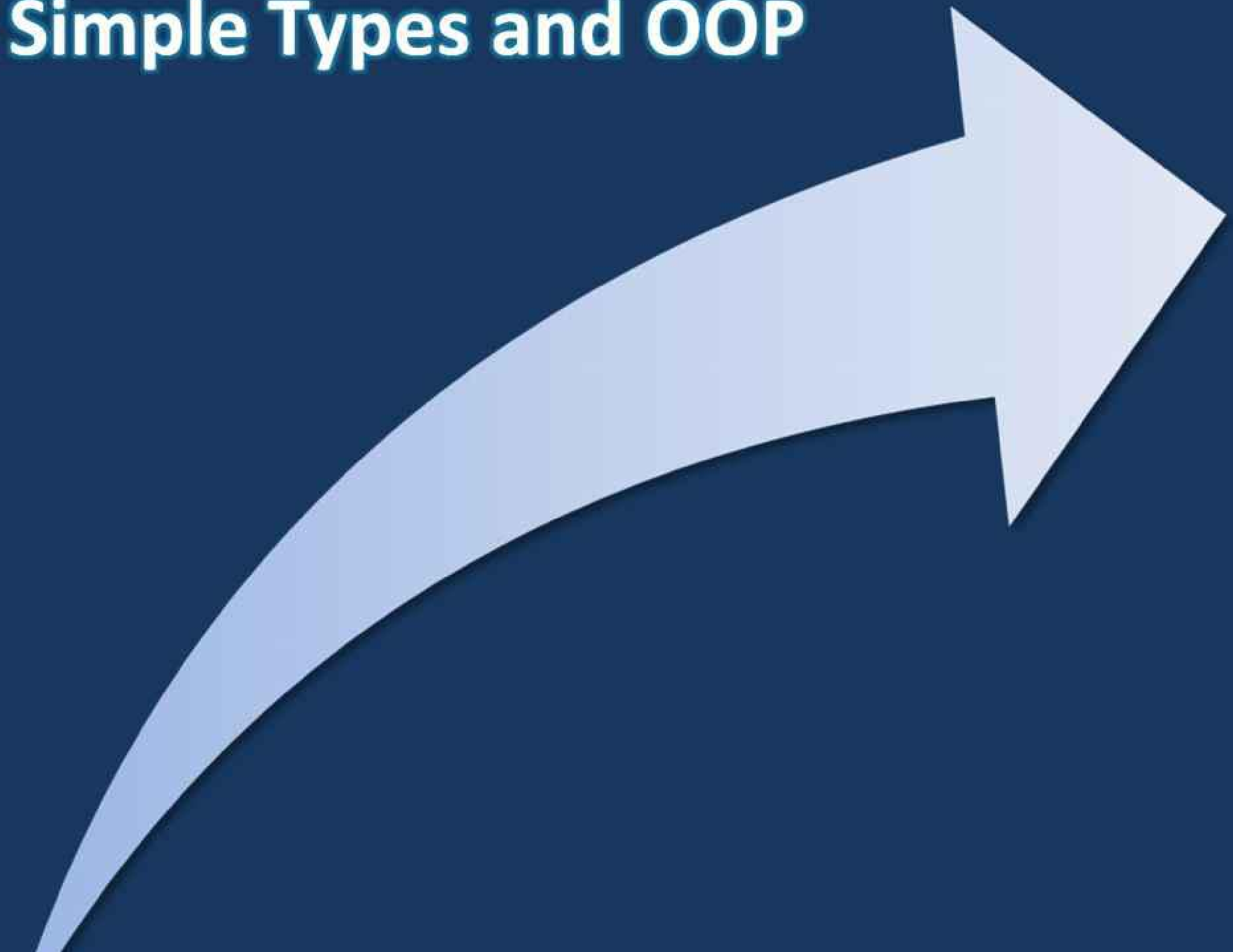
Delphi XE2 Foundations

Part 1: Language Basics, Simple Types and OOP



Chris Rolliston

Delphi XE2 Foundations - Part 1

Copyright © 2012 Chris Rolliston

Updated 30 June 2012

Delphi is a trademark of Embarcadero Technologies. Windows is a trademark of Microsoft. Other trademarks are of the respective owners. The author makes no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accepts no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

Table of contents

General Introduction

Delphi is a complete software development environment for Windows, having its own language, libraries and 'RAD Studio' IDE. Now in its XE2 incarnation, Delphi continues to evolve. Most notably, XE2 supports writing applications for Mac OS X, an ability added in the context of numerous other extensions and improvements across recent releases.

Centred upon XE2, this book aims to provide a comprehensive introduction to the fundamentals of Delphi programming as they stand today. A contemporary focus means it has been written on the assumption you are actually using the newest version. Since a traditional strength of Delphi is how new releases rarely require old code to be rewritten, much of the text will apply to older versions too. However, the book does not explicitly highlight what will and what will not.

Conversely, a focus on fundamentals means a subject-matter of the Delphi language and wider runtime library (RTL). Topics such as the RAD Studio IDE and Visual Component Library (VCL — Delphi's longstanding graphical user interface framework) are therefore covered only incidentally. Instead, coverage spans from basic language syntax — the way Delphi code is structured and written in other words — to the RTL's support for writing multithreaded code, in which a program performs multiple operations at the same time.

If you are completely new to programming, this book is unlikely to be useful on its own. Nonetheless, it assumes little or no knowledge of Delphi specifically: taking an integrated approach, it tackles features that are new and not so new, basic and not so basic. In the process, it will describe in detail the nuts and bolts useful for almost any application you may come to write in Delphi.

About the Kindle edition of Delphi XE2 Foundations

Delphi XE2 Foundations is available in both printed and eBook versions. In eBook form, it is split into three parts; the part you are reading now is **part one**. All parts share the same ‘general introduction’, but beyond that, their content differs.

Chapter overview: part one

The first chapter of both the complete book and part one of the eBook set gives an account of the basics of the Delphi language. This details the structure of a program's source code, an overview of the typing system, and the syntax for core language elements. The later parts of the chapter also discuss the mechanics of error handling and the syntax for using pointers, a relatively advanced technique but one that still has its place if used appropriately.

Chapter two turns to consider simple types in more detail, covering numbers, enumerations and sets, along with how dates and times are handled in Delphi.

The third and fourth chapters look at classes and records, or in other words, Delphi's support for object-oriented programming (OOP). Chapter three considers the basics, before chapter four moves on to slightly more advanced topics such as metaclasses and operator overloading.

Chapter overview: part two

The first chapter of part two — chapter five of the complete book — considers string handling in Delphi. It begins with the semantics of the `string` type itself, before providing a reference for the RTL's string manipulation functionality, including Delphi's regular expressions (regex) support.

Chapter six discusses arrays, collections and custom enumerators, providing a reference for the first and second, and some worked-through examples of the third.

Chapters seven to nine look at I/O, focussing initially upon the basics of writing a console program in Delphi, before turning to the syntax for manipulating the file system (chapter seven), an in-depth discussion of streams (chapter eight), and Delphi's support for common storage formats (chapter nine).

Chapter overview: part three

Chapter ten — the first chapter of part three — introduces packages, a Delphi-specific form of DLLs (Windows) or dylibs (OS X) that provide a convenient way of modularising a monolithic executable. The bulk of this chapter works through a FireMonkey example that cross compiles between Windows and OS X.

Chapters eleven and twelve look at Delphi's support for dynamic typing and reflection — 'runtime-type information' (RTTI) in the language of Delphi itself. The RTTI section is aimed primarily as a reference, however example usage scenarios are discussed.

Chapter thirteen looks at how to interact with the native application programming interfaces (APIs). A particular focus is given to using the Delphi to Objective-C bridge for programming to OS X's 'Cocoa' API.

Chapter fourteen looks at the basics of writing and using custom dynamic libraries (DLLs on Windows, dylibs on OS X), focussing on knobbly issues like how to exchange string data in a language independent fashion. Examples are given of interacting with both Visual Basic for Applications (VBA) and C#/.NET clients.

Finally, chapter fifteen discusses multithreaded programming in Delphi, with a particular focus on providing a reference for the threading primitives provided by the RTL. This is then put to use in the final section of the book, which works through a 'futures' implementation based on a custom thread pool.

Trying out the code snippets

Unless otherwise indicated, the code snippets in this book will assume a console rather than GUI context. This is to focus upon the thing being demonstrated, avoiding the inevitable paraphernalia had by GUI-based demos.

To try any code snippet out, you will therefore need to create a console application in the IDE. If no other project is open, this can be done via the Tool Palette to the bottom right of the screen. Alternatively, you can use the main menu bar: select `File|New|Other...`, and choose `Console Application` from the `Delphi Projects` node. This will then generate a project file with contents looking like the following:

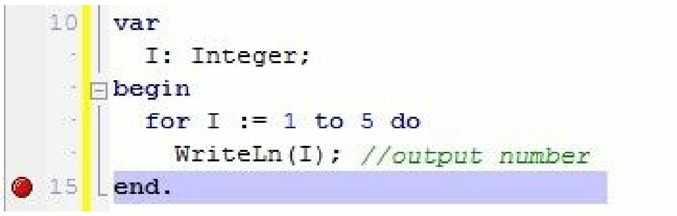
```
program Project1;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils;

begin
  try
    { TODO -oUser -cConsole Main : Insert code here }
  except
    on E: Exception do
      Writeln(E.ClassName, ': ', E.Message);
  end;
end.
```

Unless the context indicates otherwise, the `uses` to the final `end` should be overwritten with the code snippet.

By default, the console window will close when a debugged application finishes, preventing you from inspecting any output first. To avoid this, you can amend the final part of the code added to look like the following:

```
  //code...
  Write('Press ENTER to exit...',
  ReadLn;
end.
```

Alternatively, you can add a breakpoint at the end of the source code. Do this by clicking in the leftmost part of gutter beside the `end.`:



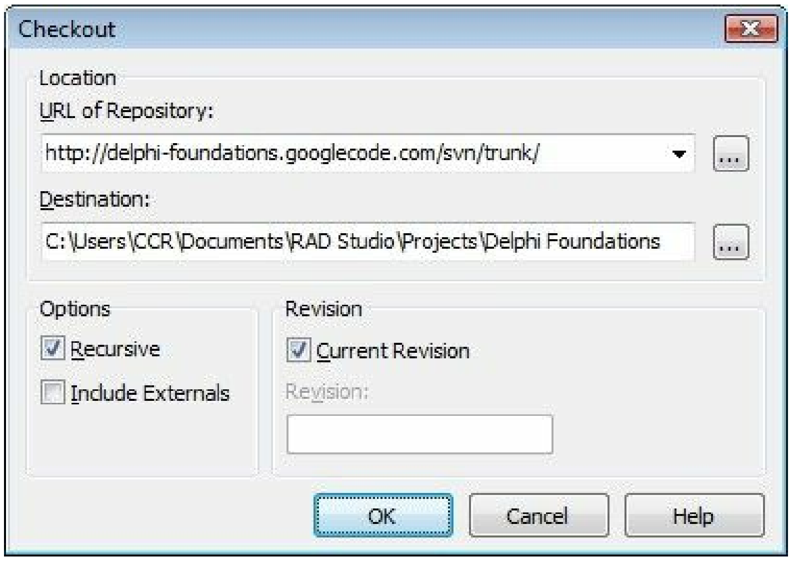
```
var
  I: Integer;
begin
  for I := 1 to 5 do
    WriteLn(I); //output number
end.
```

This will cause the IDE to regain control just before the program closes, allowing you to switch back to the console window to view the output. To allow the program to then close normally, switch back to the IDE and press either the ‘Run’ toolbar button or the F9 key, as you would have done to start the program in the first place.

Sample projects

Code for the more substantive examples in the book can be found in a Subversion repository hosted on Google Code. An easy way to download it is to use the RAD Studio IDE:

1. Choose `File|Open From Version Control...` from the main menu bar.
2. Enter `http://delphi-foundations.googlecode.com/svn/trunk/` for the URL, and a local folder of your choosing for the destination.


Checkout

Location

URL of Repository: http://delphi-foundations.googlecode.com/svn/trunk/

Destination: C:\Users\CCR\Documents\RAD Studio\Projects\Delphi Foundations

Options

[✓] Recursive

[ ] Include Externals

Revision

[✓] Current Revision

Revision:

3. On clicking OK, a new dialog should come up listing the sample projects as they get downloaded.
4. Once complete, a third dialog will display allowing you to pick a project or project group to open. It isn't crucial to pick the right one though, since you can open any of the projects as normal afterwards.

Acknowledgements

I wish to thank the following, in no particular order: John Toelken for looking over chapters 1 and 3; Kenneth Cochran and Mark Walker for spotting some typos in chapters 4 and 8 respectively; Warren Postma for some concrete suggestions concerning chapter 13; Primož Gabrijelcic for technical editing of chapter 15; and Jasper Bongertz and James Chandler for assisting with tests on physical Kindle devices.

Chris Rolliston, 14 June 2012

1. Language basics

What sort of language is Delphi?

As a language, Delphi is the combination of three things in particular: a heritage in Pascal, with its strong, static typing and preference for readability; a broad but not dogmatic implementation of the object-oriented programming (OOP) paradigm, OOP being supported alongside, rather than instead of, procedural styles; and a 'native code' compiler and environment, in contrast to the 'managed code' world of C# and Java.

A language descended from Pascal

Pascal was a much-praised language designed by Niklaus Wirth, a Swiss computer scientist, in the late 1960s and early 1970s. While Delphi is not a strict superset of Wirth's Pascal, many basic syntactical features are inherited from it:

- A preference for words over symbols, especially compared to C and C-style languages. For example, compound statements are delimited by `begin`/`end` pairs rather than curly brackets (`{}`).
- Case insensitivity — `MyVar` is the same identifier as `Myvar`, `MYVAR` and `MyVaR`, and `xor` `XOR` and `xOr` serve equally well as the 'exclusive or' operator.
- Code is laid out in a broadly 'top-down' fashion so as to enable the compiler to work in a mostly 'single pass' fashion. (Delphi isn't as strict here as the original Pascal though.)
- The existence of sets and sub-range types, strongly typed enumerations, etc.
- The syntax of basic looping constructs (`while`/`do`, `repeat`/`until`, `for`/`to`), along with that of `if`/`else` and (to an extent) `case` statements too.
- Various other syntactical bits and pieces, for example the assignment operator being a colon followed by an equals sign (`:=`), the equality operator being an equals sign on its own (`=`; the inequality operator is `<>`), and string literals using single rather than double quotation marks (`'string const'` not `"string const"`).

A mixed object oriented/procedural language

Delphi has a rich class model, similar to the ones had by Java and C#, and with a few tricks of its own to boot. However, it does not purport to be a purely OOP language. One aspect of this is how instances of basic types such as strings (i.e., pieces of textual data) and integers are not themselves objects. Another is that sub-routines ('procedures' and 'functions' in Pascal-speak), variables and constants can be declared independently of classes. Indeed, a large part of the Delphi RTL is purely procedural in nature. If you want to convert an integer to and from a string, for example, you call the freestanding `IntToStr` and `StrToInt` functions, passing the source value as a parameter:

```
uses
  System.SysUtils; //for IntToStr and StrToInt

var
  MyInt: Integer;
  MyStr: string;
begin
  MyInt := 12345;
  WriteLn('MyInt before: ', MyInt);
  MyStr := IntToStr(MyInt);
  MyInt := StrToInt(MyStr);
  WriteLn('MyInt after: ', MyInt);
end.
```

Put simply, while most Delphi code you write yourself is likely to be class based, or at least, partly class based, there is no requirement to adopt a completely OOP approach if you don't want to.

A native code compiler

In contemporary terms, Delphi is a 'native code' development tool, contrasting with the 'managed code' nature of environments such as Java and .NET. 'Native code' can mean various things, but it pertains to three features in particular:

- Unlike in a managed code system, the Delphi compiler compiles direct to machine code instead of an intermediary 'bytecode'.
- Delphi programs do not run inside a 'virtual machine' — instead, they run on top of core operating system APIs and

nothing else.

- Low level techniques such as pointer manipulation and integrating assembly code can be used.

A further consequence is that programming in Delphi is inherently ‘closer to the metal’ than programming in managed code environments, let alone scripting languages like JavaScript. This does have some costs; most notably, memory management sometimes must be explicitly coded, since there is no global ‘garbage collector’ that automatic frees up memory. However, memory management is not exactly difficult either, and you rarely (if ever) *have* to use lower-level constructs. Rather, they are there for as and when you want to make use them.

The basic structure of a Delphi program

The 'do nothing' program

At its simplest, the code for a Delphi program is composed of a single source file, which should have a DPR (Delphi PRoject) extension. A do-nothing program looks like the following:

```
program ProgramName;
begin
end.
```

The name of the source file should match the name of the application, as it appears following the `program` keyword. In the present example, the file should therefore be called `ProgramName.dpr`, producing an executable file called `ProgramName.exe` on Windows and `ProgramName` when targeting OS X.

If your edition of Delphi includes the command line compiler (and all but the low end 'Starter' edition do), you could create a project file in Notepad and compile it from a command prompt. Much more likely is the case of creating it in the IDE however. Do this, and a series of other files will be created too. Only the DPR (DPK in the case of a 'package') and any PAS files contain source code; the others will be configuration files of various kinds, most notably the DPROJ, an MSBuild makefile.

When creating a new project in the IDE, you will be presented with various options. Choose 'Console Application', and the following boilerplate code is produced:

```
program Project1;

{$APPTYPE CONSOLE}

uses
  System.SysUtils;

begin
  try
    { TODO -oUser -cConsole Main : Insert code here }
  except
    on E: Exception do
      Writeln(E.ClassName, ': ', E.Message);
  end;
end.
```

Compared to the bare-bones 'do nothing' program, there are a few more things going on here:

- The `{$APPTYPE CONSOLE}` line is a 'compiler directive', telling the Delphi compiler (or more exactly, the Delphi linker) that it is a command-line application.
- A 'uses' clause is defined, which imports the public symbols of the `System.SysUtils` 'unit'. If the uses clause were to be removed, the project wouldn't compile since the `Exception` type is exported by `System.SysUtils`.
- The code between the `begin` and `end` define what the program actually does, which in this case still isn't very much! Specifically, only a custom 'default exception handler' is defined. This makes clear that any 'exception' (i.e., runtime error) will ultimately be trapped and not escape the program. Because of this, the application won't ever 'crash' from the point of view of the operating system. (As an aside: `begin`/`end` pairs appear a lot in Delphi. Technically they constitute a 'compound statement', i.e. a series of statements where only one would be allowed otherwise.)
- The curly bracket pair `{}` defines a 'comment', i.e., some descriptive text for the programmer's benefit but ignored by the compiler.

More comments

Alongside curly brackets, Delphi also supports two other comment styles: `(* *)` and `//`. The `{ }` and `(* *)` styles allow for 'multiline' comments, i.e. where the comment text may span more than one line. In contrast, a pair of forward slashes means the rest of the current line gets interpreted as a comment:

```
(* Crucial procedure that does the following:
   - Nothing as yet.
   - Something in the future                 *)

procedure Foo; //single line comment
begin
end;
```

In the IDE, you can press `Ctrl+/` to toggle single line comment markers for the selected line or lines.

According to the rules of Standard Pascal, `(*` should be interchangeable with `{`, and `*)` interchangeable with `}`. However, Delphi treats them as different in order to enable nested multiline comments:

```
(* Outer comment.
   { Inner comment.
     Lovely.}        *)
```

Technically, a compiler directive added to the source code is also a ‘comment’, or more exactly, part of one. The following therefore both sets a compiler directive and provides some descriptive text (ignored by the compiler) at the same time:

```
{$APPTYPE CONSOLE Text that means nothing in particular...}
```

This does too, though using the double slash comment style won’t work:

```
(*$APPTYPE CONSOLE Adding a comment here is potentially confusing though. *)
```

Significantly, adding just a single space in between the comment begin marker and the dollar sign (`$`) will turn the directive back into being just a comment. Luckily, the IDE’s default formatting settings will indicate this, with the editor italicising a mere comment where it won’t with compiler directive.

Doing things

As is traditional, here is the source for a 'hello world' application in Delphi, specifically a command-line (console) version. For clarity, I have removed the default exception handler from the IDE's boilerplate code, since it isn't strictly necessary:

```
program Project1;

{$APPTYPE CONSOLE}

begin
  Write('Hello World!');
  ReadLn;
end.
```

Between the `begin` and `end` are two 'statements', the first invoking the `Write` standard procedure to output `Hello world` and the second calling `ReadLn` to wait for the `ENTER` key to be pressed. Since both `Write` and `ReadLn` are 'intrinsic routines' defined by the language, no `uses` clause is required.

Technically, statements in Delphi are *separated* rather than *terminated* by semi-colons. The above example therefore formally contains *three* statements, the final one (defined between the last semi-colon and the `end`) being empty. Since the compiler generates no code for an empty however, the fact one formally exists has no practical consequence though, and indeed, many Delphi programmers would actually expect it!

Whitespace

Something else that is harmless from the compiler's point of view is how whitespace (particularly line breaks) are used between statements:

```
program Project1;

{$APPTYPE CONSOLE}

begin
  WriteLn('1st statement'); WriteLn('2nd statement');
end.
```

Nonetheless, it is good practice to use one line per statement. Aside from being more readable, it allows you to set a breakpoint specific to a particular statement, which aids debugging.

An exception to the compiler's liberality about line breaks is with 'string literals' though (in the snippet just presented, `'1st statement'` and `'2nd statement'` are both examples of string literals). The following code does not therefore compile:

```
WriteLn('Multiple
  lines');
```

If your intention was merely to break up the string literal in the code editor, the following would fix things:

```
WriteLn('Multiple ' +
  'lines');
```

Alternatively, if your intention was to include a line break in the literal itself, use two literals and add `SLineBreak`, a global constant, in between them:

```
WriteLn('Multiple' + SLineBreak + 'lines');
```

To provide a slightly more WYSIWYG look to things, you can break it into multiple lines:

```
WriteLn('Multiple' + SLineBreak +
        'lines');
```

Doing a few more things

For completeness' sake, here's a non-console 'hello world' variant:

```
program Project1;

uses
  Vcl.Dialogs;

begin
  ShowMessage('Hello World!');
end.
```

Since not requiring console output is the default, we can just leave out the `$APPTYPE` compiler directive, notwithstanding the fact using `{$APPTYPE GUI}` would make non-console output explicit. However, we definitely need to import the `Vcl.Dialogs` unit (part of the Visual Component Library), since displaying dialog boxes is a library rather than a language feature.

Going a little beyond 'hello world', here's a slightly more complicated program, though one that is still self-contained in a single code file. The comments describe what each bit of the code is for, what it does, or what it is:

```
program Project1;

{ A 'uses' clause enables using things declared by the
  specified 'units'; here, we use the Vcl.Dialogs unit }
uses
  Vcl.Dialogs;

{ Declare and implement a custom sub-routine that reverses
  the given string (i.e., piece of textual data). E.g., if
  'hello' is passed in, 'olleh' will be returned }
function ReverseString(const S: string): string;
var                                    //Local variable block
  I, Len: Integer;                     //Local variables
begin
  Len := Length(S);                    //Assignment statement
  SetLength(Result, Len);              //Procedure call
  for I := 1 to Len do                 //'For/to' loop
    Result[Len - I + 1] := S[I];       //Updates the function result
end;

{ Another custom sub-routine; as it doesn't return anything,
  it is a 'procedure' rather than a 'function' }
procedure ShowInfo(const S: string);
begin
  { Call the MessageDlg routine from Vcl.Dialogs, which takes a
    string, a value from an 'enumeration', a 'set' & an integer }
  MessageDlg(S, mtInformation, [mbOK], 0);
end;

const                                  //Constant block
  MaxBlankEntries = 2;                 //'True constant' declared
var                                    //Variable block
  Counter: Integer;                    //Variable declaration
  TextToRev, RevText: string;          //Two more of the same type
begin
  Counter := 0;                        //Initialise the counter.
  { InputQuery shows a simple input dialog, returning True when
    the user OKed it or False if they cancelled. Here, we call
    InputQuery in a loop, repeatedly asking for a new string to
    reverse then reversing it, asking for another string etc. }
  while InputQuery('Example Program', 'Enter text to reverse:',
    TextToRev) do
  begin
    if TextToRev = '' then             //Was nothing entered?
    begin
      Inc(Counter);                    //Adds 1 to the current value
      if Counter = MaxBlankEntries then
      begin
        ShowInfo('You must have had enough!');
        Break;                         //Break out of the loop
      end;
      Continue;                        //Continue to next iteration
    end;
    { Call our custom function }
    RevText := ReverseString(TextToRev);
    { Call our custom procedure, composing argument in place }
    ShowInfo('"' + TextToRev + '" becomes "' + RevText + '"');
    Counter := 0;                     //Reset counter for next time
    TextToRev := '';                  //Reset string variable
  end;
end.
```

This demonstrates a few more syntactical elements:

- Custom sub-routines, specifically one that returns a value (a 'function') and one that doesn't (a 'procedure'). Each has its own 'identifier' (`ReverseString` and `ShowInfo`).
- The declaration and use of 'variables', which again have their own identifiers (`I`, `Len`, `Counter`, etc.). Notice they are declared in a `var` block *before* the executable statements they pertain to.

- The fact variables, sub-routine parameters and function results all have an explicit ‘type’. In this example, that type is either `string` or `Integer`, but many more are possible.
- The declaration and use of a ‘constant’, i.e. a fixed value that is given its own identifier.
- An `if` statement and two looping constructs (`for`/`to` and `while`).

Identifiers

The names you give to the things in your program's source code — its variables, types, sub-routines, and so on — must take the form of valid identifiers. In Delphi, this means a name that is up to 255 characters in length, that does not include any spaces or symbols used by the language itself (e.g. full stops/periods, commas, colons, etc.), and that does not start with a number. Adhering to these rules, the following declarations all compile:

```
var
  IsOK: Boolean;
  _also_ok_but_not_idiomatic: string;
  ΜαγικόΑριθμό: Integer;
  _999: Char;
  _: Variant;
```

Other things being equal, you should use identifiers like the first example though, i.e. names composed solely of Latin letters. Quite simply, just because you *can* enter inscrutable code contests with Delphi code doesn't mean you should!

Reserved words

A further restriction on your choice of identifiers is that you cannot naïvely use any of the Delphi language's 'reserved words'. In alphabetical order, these amount to the following:

```
and, array, as, asm, begin, case, class, const, constructor,
destructor, dispinterface, div, do, downto, else, end, except,
exports, file, finalization, finally, for, function, goto, if,
implementation, in, inherited, initialization, inline, interface,
is, label, library, mod, nil, not, object, of, or, out, packed,
procedure, program, property, raise, record, repeat,
resourcestring, set, shl, shr, string, then, threadvar, to, try,
type, unit, until, uses, var, while, with, xor
```

If you really must reuse a reserved word, you are allowed to do so if you prefix it with an ampersand:

```
var
  &File: string; //declare a string variable called File
```

In general this is not advisable however — just accept certain identifiers are 'off limits'.

Casing and naming conventions

By convention, reserved words used for their reserved purpose are written in all lower case. Otherwise, so-called Pascal case is the norm (`FullName` not `full_name` or `fullName` etc.). An additional one letter prefix is also commonly used for identifiers of certain kinds of thing:

- Most type names are given a 'T' prefix (hence, `TObject`, `TFileMode` etc.); simple types like `Integer` aren't though.
- Exception classes, which categorise runtime errors, are prefixed with an 'E' (e.g. `EInvalidCast`).
- Pointer type aliases are prefixed with a 'P' (e.g. `PInteger` — a pointer to an `Integer` value).
- Object fields are prefixed with an 'F' (e.g. `FCaption`).
- Procedure or function parameters are sometimes prefixed with an 'A' (for 'argument').
- Local variables are occasionally prefixed with an 'L'.

However, no naming convention is enforced by the language, and indeed, amongst the prefixes, only the 'T', 'E', 'P' and 'F' ones tend to be consistently used in practice.

Even so, if you wish to write code that will be easily understood by other Delphi programmers, using these conventions is a good idea. For more formatting suggestions, consult Charlie Calvert's 'Object Pascal Style Guide', as found on the Embarcadero Developer Network (`http://edn.embarcadero.com/article/10280`). This document is quite old, and as a result, doesn't cover newer language features like parameterised types. Nonetheless, it continues to have authority due to its aim of making explicit the best stylistic practices of the core RTL and VCL sources, which continue to set the standard for what good Delphi code looks like.

Typing in Delphi

Delphi is a strongly and statically typed language: 'strongly' typed, since almost every piece of data as used must have a specific 'type' or defined meaning and form; and 'statically' typed, since this typing is generally enforced at compile time.

Common types

Basic types defined by the language include `Boolean`, for true/false values; `Integer`, for signed whole numbers; `Double`, for floating point numbers (floats are numbers that may include a fractional element); `string`, for textual data; and `Char`, for single characters. When something that has or returns data is declared, the name of the data type usually follows shortly after the name of the thing itself:

```
var
  MyBool: Boolean;
  MyInt: Integer;
  MyFloat: Double;
  MyStr: string;
  MyChar: Char;
begin
  MyBool := True;
  MyInt := 45;
  MyFloat := 123.456;
  MyStr := 'Example';
  MyChar := 'z';
```

The ability to directly assign variables of one type to another is deliberately limited. Thus, while you can directly assign an integer to a float, or a character to a string, you cannot go the other way without doing something else. In the numbers case, this 'something' is to explicitly round or truncate; in the case of assigning a string to a character, it is specify *which* character to assign, which you do by 'indexing' the string:

```
MyInt := Round(MyFloat); //round to the nearest whole number
MyInt := Trunc(MyFloat); //just truncate if it contains a decimal
MyChar := MyStr[2];      //get the second character
```

Similarly, putting a number into a string, or converting a string representation of a number into an actual number, requires a function call:

```
MyInt := 123
MyStr := IntToStr(MyInt);
MyStr := '456.789';
MyFloat := StrToFloat(MyStr);
```

The principle behind this is to force you to make your intentions explicit when the compiler cannot know for sure that a type conversion will be 'lossless'. For example, while a floating point variable may be in practice only have whole numbers assigned to it, this will be unknown at the time the compiler converts your source code into machine instructions. While it can seem a chore initially, strict typing makes code more maintainable in the long run, since a range of potential errors are either caught much earlier than otherwise, or have their cause made much more obvious when they do occur.

Variants and pointers

Two partial exceptions to Delphi's otherwise strongly typed approach are 'variants' and 'pointers'. A variant can be assigned different sorts of data, dynamically adapting its 'real' type in step and performing automatic conversions as necessary:

```
var
  V: Variant;
begin
  V := '123';       //put in a string
  V := V + 1111.56; //1234.56 (i.e., a float)
```

In practice, variants are a feature for specific occasions (in particular, higher level COM programming on Windows), rather than a general-purpose alternative to proper typing.

This conclusion also holds with pointers, a low-level feature that requires some expertise to use properly. A 'pointer' is so called because it 'points' to a specific place in memory: a pointer, then, contains a memory address *to* a piece of data, rather than being a piece of data as such. While coding in Delphi always involves working with things that are on one level concealed pointers, dealing with pointers proper is never actually required, in contrast to other native code languages like C and C++.

Custom types

Beyond the built-in types, you can define your own using a variety of techniques. In each case, a custom type is declared inside a type block:

```
type
  TMyCustomType = //...
```

Type aliases map custom identifiers to existing types:

```
type
  TSuperID = Integer;
  TStringIntegerDictionary = TDictionary<string, Integer>;
```

Sub-range types restrict 'ordinal' values like integers to be within a certain range:

```
type
  TOutOfTen = 1..10;
```

Enumerated types define groups of ordered identifiers:

```
type
  TDayOfWeek = (Mon, Tues, Wed, Thurs, Fri, Sat, Sun);

var
  Day: TDayOfWeek;
begin
  Day := Sat;
```

Set types are based on ordinal types (typically enumerations), and have instances that include one or more values of the base type. Each element of the base type can appear only once, and there is no inherent order to the values included:

```
type
  TDayOfWeekSet = set of TDayOfWeek;

var
  Days: TDayOfWeekSet;
begin
  Days := [Mon, Wed]; //Days now contains Mon and Wed
```

Array types define continuous sequences of data that are accessed by an index. Delphi supports both 'static' arrays, whose bounds are determined at compile time, and 'dynamic' arrays, whose bounds are set (and possibly reset) at runtime:

```
type
  TFixedNameArray = array[1..5] of string; //static array type
  TFlexibleNameArray = array of string;    //dynamic array type

var
  Names: TFixedNameArray;
  OtherNames: TFlexibleNameArray;
begin
  Names[1] := 'Bob';
  Names[2] := 'Bill';
  SetLength(OtherNames, 4);
  OtherNames[0] := 'Ben'
```

Record types are similar to arrays in that they group pieces of data together. Instead of items being accessed through an index however, they have their own identifiers:

```
type
  TPersonRec = record
    Title, Forename, Surname: string;
    Age: Integer;
  end;

var
  Rec: TPersonRec;
begin
  Rec.Title := 'Mr';
  Rec.Forename := 'Joseph';
  Rec.Surname := 'Bloggs';
  Rec.Age := 36;
```

Records support basic OOP functionality, in particular encapsulation (i.e., preventing state — internal fields — from being open to direct manipulation by calling code) and the ability to have 'methods' (i.e. procedures or functions attached to the type):

```
type
  TSimpleRecordObject = record
  strict private                      //Encapsulate what follows
    FData: string;
  public                              //Define accessors:
    procedure AddToData(S: string);   //- Method
    property Data: string read FFata; //- Read-only property
  end;

procedure TSimpleRecordObject.AddToData(S: string);
begin
  FData := FData + S;
end;
```

Classes are Delphi's primary type for object-oriented programming. Beyond what records offer, they add support for inheritance, 'virtual' methods, and the ability to implement one or more 'interfaces' (all these things will be explained in chapters 3 and 4):

```
type
  { Define an 'interface', or way to refer interchangeably to
    different classes that implement a given group of methods }
  IFriendlyObject = interface
    procedure SayHello(S: string);
  end;

  { Inherit from TInterfacedObject & implement IFriendlyObject }
  TSimpleClassBasedObject = class(TInterfacedObject, IFriendlyObject)
  strict private
    FHelloText: string;
  protected
    procedure SetHelloText(const ANewText: string); virtual;
  public
    procedure SayHello(S: string);
    property HelloText: string read FHelloText write SetHelloText;
  end;

  { Inherit from TSimpleClassBasedObject & 'override' (i.e.,
    change the behaviour of) SetHelloText }
  TMoreComplicatedObject = class(TSimpleClassBasedObject)
  protected
    procedure SetHelloText(const ANewText: string); override;
  end;

procedure TSimpleClassBasedObject.SetHelloText(const ANewText: string);
begin
  FHelloText := ANewText;
end;

//and so on for the other method bodies...
```

Metaclasses (sometimes called 'class references') are an abstraction away from classes. Specifically, each class has an implicit metaclass that can itself have methods (including virtual methods) and properties. These are declared as 'class' members within the normal class declaration:

```
type
  TFoo = class
    class function Description: string; virtual;
  end;

  TFooClass = class of TFoo; //make metaclass type explicit
```

Class methods typically describe the class in some form, or more generally, offer functionality that does not require the class to be 'instantiated' first (i.e., an object created from it). The ability to declare virtual 'constructors' also provides an easy way to choose what particular class to instantiate at runtime without having to write parallel code paths, a feature Delphi's two visual frameworks (the traditional Visual Component Library [VCL] and the cross platform, graphically rich FireMonkey) rely on heavily.

Types of types

In order to get going, just knowing what types are commonly used may be enough. Nevertheless, it will soon become important to understand how types in Delphi fall into various categories, with different sorts of type having their own distinctive behaviour.

Ordinal types

Ordinals are types that have an integral representation internally, namely integers themselves and enumerations. Only ordinal types can be used for the selector in a 'case' statement, or the index of a static array. They also share a number of intrinsic routines such as `Low`, `High`, `Inc`, `Dec`, `Pred` and `Succ`. `Low` returns the first element of the enumeration, `High` the last, `Inc` increments the given value, `Dec` decrements it, `Pred` is a function that returns the previous ('predecessor') value, and `Succ` returns the next ('successor') value:

```
type
  TNameEnum = (Bob, Jane, Joe, Tim);

var
  I1, I2: Integer;
  N1, N2: TNameEnum;
begin
  I1 := 1;
  Inc(I1, 3);            //increment I1 by 3, making it 4
  I2 := Succ(I1);        //set I2 to successor value of I1 (5)
  N1 := High(TNameEnum); //assigns Tim
  N2 := Pred(TNameEnum); //sets N2 to Joe, i.e. value before Tim
```

Value types

Value types include all ordinal and floating point types, along with static arrays, sets, records and variants. When you assign one instance of a value type to another, the actual data is copied. As a result, subsequently changing the data held by the first has no effect on the second:

```
var
  A, B: Integer;          //Declare two integers
begin
  A := 99;                //Initialise first integer
  B := A;                 //Assign to second; value is copied
  A := 11;                //Change first integer
  WriteLn('A = ', A);     //output: A = 11
  WriteLn('B = ', B);     //output: B = 99
end.
```

Reference types

An instance of a reference type is an implicit pointer, merely referencing its data rather than being the data itself. Technically, this means the memory for a reference type is always allocated dynamically on the 'heap', in contrast to how the memory for a value type is allocated on the 'stack'.

Reference types include dynamic arrays, classes, interfaces, and (with caveats) strings. In practical terms, being a reference type means assigning one instance to another only sets up the second to reference the original data, not copy the data over:

```
var
  A, B: TArray<Integer>;           //Declare two dynamic arrays
begin
  A := TArray<Integer>.Create(99); //Initialise with one element
  B := A;                          //Assign reference
  A[0] := 11;                      //Change the element
  WriteLn('A[0] = ', A[0]);        //output: A[0] = 11
  WriteLn('B[0] = ', B[0]);        //output: B[0] = 11
end.
```

A further feature of reference types is that their instances may be assigned (and tested for) the special value `nil`, which means 'no data' (in the case of dynamic arrays, this is equivalent to having zero elements). If you don't initialise variables with reference types explicitly, this is also the value they will be set to implicitly.

Strings make for a joker in the pack however, since they are reference types with apparently value type behaviour. Thus, when two or more string variables point to the same data, assigning something else to the first will cause the implicit association with the other references to be broken, as if the original data was never shared in the first place. Furthermore, you cannot explicitly assign `nil` to a string. Instead, you must assign an empty string (`''`), which will cause nil to be assigned internally. The equation of nil with an empty string means you never have to test for nil before comparing two string values.

Managed types

A managed type is one that has special handling by the compiler. This 'special handling' will generally involve memory management, the compiler handling the dynamic allocation, reallocation and deallocation of a managed type's memory for you. Under the bonnet, this management usually takes the form of reference counting, though it doesn't have to be.

While most managed types are reference types, and most reference types are managed types, neither side of the equation necessarily holds. In particular, classes (which are reference types) are not 'managed', but variants (which are value types) are. Furthermore, any record or static array that contains at least one item with a managed type will effectively become an honorary managed type itself. This is to ensure the managed item or items are initialised and finalised correctly.

Pointer types

A pointer is a variable that refers to a particular memory location. Pointers in Delphi come in two main forms. The first are 'untyped' pointers, denoted by the Pointer type. These do not claim to point to anything in particular — they just point to a particular place in memory. As such, you can't do much with them aside from testing whether one points to the same address as a second:

```
var
  I1, I2: Integer;
  P1, P2: Pointer;
begin
  P1 := @I1;
  P2 := @I1;
  WriteLn(P1 = P2); //output: TRUE
  P2 := @I2;
  WriteLn(P1 = P2); //output: FALSE
```

As shown here, the @ symbol 'references' the thing to its immediate right, setting the pointer to point to it.

The other sort of pointers are 'typed' pointers, which purport to point to an instance of a certain type (only purport, because it is up to you to ensure that really is the case). Predefined typed pointer types include PByte, a pointer to a Byte, PChar, a pointer to a Char, and PInteger, a pointer to an Integer. When a typed pointer is used, the ^ symbol 'dereferences' the pointer:

```
var
  I: Integer;
  P: PInteger;
begin
  I := 100;
  WriteLn(I); //output: 100
  P := @I;
  P^ := 200;
  WriteLn(I); //output: 200
```

The ^ symbol is also used to declare a new sort of typed pointer in the first place. A typed pointer type can either be declared 'inline' with the variable, or more idiomatically, as an explicit type:

```
type
  TMyRec = record
    A, B: Integer;
  end;

  PMyRec = ^TMyRec; //declare an explicit typed pointer type

var
  MyPtr: ^TMyRec;   //declare type inline with variable
```

While it is only a convention, experienced Delphi programmers will expect pointer types to be named by taking the name of the base type, removing the T prefix (if it exists) and prepending a P one. Thus, the predefined pointer type for Integer is PInteger, and the pointer type for our TMyRec record was named PMyRec.

Generic types

The term 'generic type' has two distinct meanings in Delphi. In the first, all of Integer, Real, string and Char are said to be 'generic' because their precise implementation may change between compiler versions. In other words, each maps to a specific 'fundamental' type but not necessarily the same fundamental type for every version of Delphi. For example, Integer currently maps to the 32 bit signed integer type (LongInt), though in the past it mapped to a 16 bit one (SmallInt) and in the future might map to a 64 bit one (Int64).

In another context, ‘generic’ types are types whose instances are themselves types (this use of the word comes from Java and C#). Unlike ‘generic’ types in the first sense, these are not defined by the compiler itself. Rather, they are defined in Delphi code, which you can spot by their use of angle brackets:

```
type
  TExampleGeneric<T> = record
    Data: T;
  end;
```

Between the angle brackets are placeholders for types used by one or more fields, properties or methods. To be used, a generic must be ‘instantiated’ by filling out the specific type or types to use:

```
var
  Rec1: TExampleGeneric<Integer>;
  Rec2: TExampleGeneric<string>;
begin
  Rec1.Data := 123;
  Rec2.Data := 'Hello generics world!';
```

Variables and constants

A 'variable' is an assignable piece of data. Every variable must have a type. Variables also come in three main forms: 'global variables', 'local variables' and instance 'fields'. Global variables are simply those that are declared outside of sub-routines and objects; local variables, those that are declared within a sub-routine; and fields, those that are declared within a class or record type:

```
var
  GlobalVariable: string;

procedure MyProc;
var
  LocalVariable: Integer;
begin
  for LocalVariable := 1 to 3 do
    WriteLn(LocalVariable);
end;

type
  TMyObject = class
    Field: Boolean;
  end;

  TMyRecord = record
    AnotherField: Real;
  end;

begin
end.
```

Of the three, only global variables can be initialised at the point of their declaration:

```
var
  GlobalVariable: string = 'Initial value';
```

Global variables that are not explicitly initialised are implicitly 'zero-initialised' instead. This means numerical values are set to `0`, `nil`'able types set to `nil`, and strings set to empty (`''`). Also zero-initialised are the instance fields of a class (though *not* those of a record), along with all variables — including local ones — that have a managed type. Local variables with non-managed types won't be automatically initialised with anything though.

For example, consider the following code snippet:

```
procedure Foo;
var
  I: Integer;
  Obj: TObject;
  S: string;
begin
  //do stuff
```

Since `Integer` and `TObject` are not managed types, and `I` and `Obj` are local variables, they will be undefined when the routine begins executing (maybe they will be `0` and `nil` respectively, maybe they won't!). However, because `S` has a managed type, it will be automatically initialised to an empty string.

Constants — 'true' vs. 'typed'

Constants come in two main forms, 'true constants' and 'typed constants'. Both forms are declared in a `const` block (possibly the same `const` block), and both forms have their values determined by a 'constant expression'. A constant expression may mean any of the following:

- A literal, like 42, 1.23 or `'string'`.
- Another true constant.
- A compiler-magic function that resolves at compile-time. The most prominent example of this is `SizeOf`, which returns the size in bytes of an instance of a specified type or variable.
- A computation that uses only built-in operators, literals, other true constants, and/or the aforementioned compiler magic functions:

```
const
  TrueConstFromLiteral = 42;
  TypedConstFromLiteral: Integer = 99;
  ComputedTrueConst = TrueConstFromLiteral + 89;
  ComputedTypedConst: Double = TrueConstFromLiteral / 2.1;
  TwoPointerSizes = SizeOf(Pointer) * 2;
```

The general form of a typed constant declaration is therefore as thus:

```
ConstName: TypeName = ConstExpression;
```

Notice how this is very similar to the declaration of an initialised global variable. The similarity is in fact no accident, since typed constants are actually variables that the compiler imposes read-only semantics on. This is why you are not able to use a typed constant in a constant expression — a typed constant isn't actually a 'constant' under the hood!

The declaration of a true constant, in contrast, can take a couple of forms:

```
ConstName = ConstExpression;
ConstName = TypeName(ConstExpression);
```

With the first form, the type of the constant is inferred by the compiler. A string expression, as you would expect, resolves to a `string` constant, unless the length is one character exactly, with which a `Char` constant is produced instead. In the case of numeric expressions, in contrast, there are decisions to be made, and the compiler does the following: in the case of an expression that results in a whole number, the type becomes the smallest sort of integer that can hold the value concerned. For example, the literal `33` would result in a `Byte` constant. In the case of a floating point value, however, the *largest* floating point type is used, which in 32 bit Delphi is `Extended` rather than `Real`/`Double`.

In the case of strings and integers, if you don't want to be at the whim of the compiler's inferences, you can instead use the second form and give your true constant a type explicitly, using what looks like a hard cast (a 'hard cast' is when you instruct the compiler to interpret data typed to one thing as typed to another, no questions asked):

```
const
  MyStr = string(':');
  MyInt64 = Int64(99);
```

Doing this is generally not necessary however; in particular, there is no problem assigning a smaller-sized integer to a larger one.

'Writeable typed constants' (static variables)

A traditional feature of C-style languages and even Visual Basic is 'static variables'. These are a special sort of local variable that keep their value between calls. By default, Delphi does not have such a beast, but with one small compiler setting, it appears in the form of 'writeable typed constants'.

The setting concerned is the `$WRITEABLECONST` directive (`$J` for short):

```
{$WRITEABLECONST ON}    //or {$J+}
procedure Foo;
const
  Counter: Integer = 0; //set the initial value
begin
  Inc(Counter);
  WriteLn(Counter);
end;
{$WRITEABLECONST OFF}   //or {$J-}

begin
  Foo; //outputs 1
  Foo; //outputs 2
  Foo; //outputs 3
end.
```

You can enable writeable typed constants globally under the 'Compiler' build configuration settings (they're called 'assignable typed constants' in the IDE's user interface). Doing that — or even using writeable typed constants at all — tends to be frowned upon nowadays though, not for the least because the phrase 'writeable constant' sounds oxymoronic, a bit like referring to a person as an 'ethical shyster'. On the other hand, disabling this functionality doesn't magically turn typed constants into true ones — all you're doing is removing an occasionally useful feature that could be better named.

Looping constructs

While and repeat/until loops

A `while` statement repeatedly performs some code whilst a Boolean expression returns `True`:

```
while BooleanExpression = True do
  Statement
```

If the expression returns `False` the first time around, the code inside the loop is never performed:

```
var
  BoolVal: Boolean;
begin
  BoolVal := False;
  while BoolVar do
    WriteLn('this line never executes');
```

In contrast, a `repeat`/`until` statement repeatedly performs one or more statements until a Boolean expression returns `True`:

```
repeat
  Statement(s)
until BooleanExpression = False
```

Since the test is performed at the end of a `repeat`/`until` loop, the code within the loop will iterate at least once:

```
var
  BoolVal: Boolean;
begin
  BoolVal := False;
  repeat
    WriteLn('this line will execute once');
  until not BoolVal;
```

Another difference with `while` loops is that where any number of statements can be placed between the `repeat` and `until`, only one can appear after the `do`:

```
Counter := 4;
while Counter > 0 do
  WriteLn(Counter);
  Dec(Counter);
```

The loop here is endless since the counter is decremented only *after* the loop, not *inside* of it. The fix is simple however — wrap the statements that should be inside the loop in a `begin`/`end` pair:

```
var
  Counter: Integer;
begin
  Counter := 0;
  repeat
    Inc(Counter);
    WriteLn(Counter);  //no begin/end needed here...
  until (Counter = 4);

  while Counter > 0 do
  begin
    WriteLn(Counter);
    Dec(Counter);      //...but is here
  end;
```

For/in loops

A `for`/`in` loop (frequently called a ‘for each’ loop in other languages) enumerates something:

```
for Element in Enumerable do
  Statement
```

Here, Element must be an appropriately typed local variable, and Enumerable must be an expression returning one of the following:

- An array
- A set
- A string
- A class or record instance that implements the enumerable pattern (what exactly that means will be discussed in

chapter 6). All standard collection classes (`TList`, `TDictionary` etc.) do this, along with VCL classes like `TComponent` and `TMenuItem`.

A compiler error will result if you attempt to use for/in where none of these conditions obtains.

As with `while` loops, if you want the loop body to contain more than one statement, you will need to wrap them inside a `begin`/`end` pair:

```
const
  MyString = 'I am feeling loopy today';
var
  Ch: Char;
begin
  for Ch in MyString do
  begin
    Write('And the next character is... ');
    WriteLn(Ch);
  end;
```

In principle, you should not care what order a for/in loop returns things. However, in practice, the convention is to return them in order *if* there is an order to return. While this won't hold for things like sets and dictionary objects, it does for arrays and strings, along with classes like `TList`:

```
uses
  System.Generics.Collections;

const
  IntArrayConst: array[1..3] of Integer = (1, 2, 3);
var
  Ch: Char;
  Int: Integer;
  List: TList<string>;
  S: string;
begin
  List := TList<string>.Create;
  try
    List.AddRange(['first', 'second', 'third']);
    for S in List do
      Writeln(S);
  finally
    List.Free;
  end;
  for Int in IntArrayConst do
    WriteLn(Int);
  for Ch in 'test' do
    Write(Ch + ' ');
end.
```

The above code produces the following output:

```
first
second
third
1
2
3
t e s t
```

For/to and for/downto loops

A `for`/`to` loop is like a `while` loop, only with a counter specified to determine how many iterations to perform. A `for`/`to` loop counts upward; a `for`/`downto` one counts downwards:

```
for Counter := LowBound to HighBound do
  Statement

for Counter := HighBound downto LowBound do
  Statement
```

Once again, if the loop body is to contain multiple statements, they must be wrapped in a `begin`/`end` pair. The counter and bounds specifiers must be either of the same type exactly, or of types that are assignment-compatible with each other. Any ordinal type is acceptable (e.g. `Integer`, `Char`, an enumeration, etc.):

```
var
  Ch: Char;
begin
```

```
  for Ch := 'A' to 'Z' do
    //do something...
```

The counter must be a local variable, however in the case of a function, the implicit `Result` variable is allowable too:

```
function IndexOfFirstColon(const S: string): Integer;
begin
  for Result := 1 to Length(S) do
    if S[Result] = ':' then Exit;
  Result := -1;
end;
```

No iteration is performed if the ‘high’ bound turns out to have a lower ordinal value than the ‘low’ bound:

```
type
  TWorkDay = (Mon, Tues, Weds, Thurs, Fri);

var
  MinDay, MaxDay: TWorkDay;
begin
  MinDay := Thurs;
  MaxDay := Tues;
  for Day := MinDay to MaxDay do
    WriteLn('this line never executes');
```

Both low and high bound are evaluated only the once, immediately prior to the first iteration. This means it is safe to call a potentially ‘weighty’ function inside the `for` statement — there is no need to put the result in a local variable first, in other words. On the other hand, this means you should be careful if using a `for` loop to clear a list:

```
procedure RemoveDummiesBuggy(List: TStrings);
var
  I: Integer;
begin
  for I := 0 to List.Count - 1 do
    if List[I] = 'dummy' then List.Delete(I);
end;
```

Say a list is passed to this routine having two items, and the first has the value `'dummy'`. On the first iteration of the loop, this item will be deleted. Since there were originally two items in the list, the loop will still continue for another iteration though... boom! A ‘list index out of range’ exception is raised. The way to avoid this is to always enumerate a list *downwards* when you may be deleting items from it:

```
procedure RemoveDummiesFixed(List: TStrings);
var
  I: Integer;
begin
  for I := List.Count - 1 downto 0 do
    if List[I] = 'dummy' then List.Delete(I);
end;
```

Jump statements — Break, Continue and Goto

In the body of a `while`, `repeat`/`until` or `for` loop, call `Break` to leave it prematurely and `Continue` to immediately move to the next iteration:

```
const
  Ints: array[1..5] of Integer = (11, 22, 33, 44, 55);
var
  Elem: Integer;
begin
  for Elem in Ints do
  begin
    if Elem mod 3 = 0 then Continue;
    WriteLn(Elem);
    if Elem > 40 then Break;
  end; //outputs 11, 22, 44
```

In the case of a `while` or `repeat`/`until` loop, using `Continue` means the normal test for whether a new iteration is to be performed is skipped. Nonetheless, both `Break` and `Continue` only work in the context of the current loop. The following code therefore outputs `1 2 1 2` rather than just `1 2`:

```
var
  I, J: Integer;
begin
  for I := 1 to 2 do
    for J := 1 to 8 do
    begin
```

```
      if J mod 3 = 0 then Break;
      Write(J, ' ');
    end;
```

In the case of a nested loop within the body of a procedure or function, `Exit` can be used if you desire to break off from the loops *and* the procedure or function itself. Otherwise, a `goto` statement may be employed:

```
var
  I, J: Integer;
label
  NextThing; //must be a valid Pascal identifier or an integer
begin
  for I := 1 to 2 do
    for J := 1 to 8 do
    begin
      if J mod 3 = 0 then goto NextThing;
      Write(J, ' ');
    end;
NextThing:
  //continue executing...
```

As shown here, the place a `goto` statement points to is marked by a 'label', the existence of which you must give prior warning of (so to speak) in a special `label` declaration section.

Unlike traditional Pascal, Delphi does not support 'interprocedural' `goto` statements, meaning you cannot use `goto` to jump from the body of one procedure or function to another. That is just as well, since finding yourself in a situation where using `goto` seems reasonable at all is frequently a sign of bad coding practices.

Conditional statements

If and if/else statements

An `if` statement takes a `Boolean` expression that, on returning `True`, will result in a further statement being called. An `if`/`else` statement is the same, only executing a third statement were the expression to return `False`:

```
if BooleanExpression = True then Statement

if BooleanExpression = True then
  Statement1
else
  Statement2
```

Notice there is no semi-colon before the `else`; in fact, adding one will result in a compiler error. If more than one statement is required for either the `if` or the `else` part, wrap the statements concerned in a `begin/end` pair:

```
procedure Foo(Param: Boolean);
begin
  if Param then
  begin
    WriteLn('Param was True');
    WriteLn('Get over it!');
  end
  else
  begin
    WriteLn('Param was False');
    WriteLn('Oh dear oh dear!');
  end;
end;
```

If you're careful, `if/else` statements can also be nested in a reasonably understandable fashion:

```
procedure NestedTest(Param1, Param2, Param3: Boolean);
begin
  if Param1 then
    //blah...
  else if Param2 then
    //blah
  else if Param3 then
    //blah
  else
    //blee
end;
```

This works because Delphi, like many other languages, matches an `else` to the nearest `if`. This may be clearer with a different indentation style:

```
procedure NestedTest(Param1, Param2, Param3: Boolean);
begin
  if Param1 then
    //blah...
  else
    if Param2 then
      //blah
    else
      if Param3 then
        //blah
      else
        //blee
end;
```

Case statements

A 'case' statement selects something to do, selecting on the basis of an expression that returns a value with an ordinal type:

```
case OrdinalExpr of
  ConstOrRange1: Statement;
  ConstOrRange2: Statement;
  ConstOrRange3: Statement;
else
  Statement(s)
end;
```

The `else` part is optional. If it doesn't exist and the ordinal expression can't be matched, then nothing happens.

Matching values cannot overlap, and must involve constant expressions only. However, ranges as well as single values may be matched using the syntax ConstExpr1..ConstExpr2. Here's an example that both uses ranges and has an `else` clause:

```
type
  THonoursClass = (hcFail, hcThird, hcLowerSecond, hcUpperSecond, hcFirst);
  TScore = 0..100;

function ScoreToClass(Score: TScore): THonoursClass;
begin
  case Score of
    40..49: Result := hcThird;
    50..59: Result := hcLowerSecond;
    60..69: Result := hcUpperSecond;
    70..100: Result := hcFirst;
  else
    Result := hcFail;
  end;
end;
```

Since case statements only work with ordinal types, string case statements are unfortunately not possible:

```
procedure WillNotCompile(const Value: string);
begin
  case Value of
    'Bob': WriteLn('Very good at building things');
    'Jack': WriteLn('Handy at many things, excellent at none');
  end;
end;
```

Procedures and functions

The most basic way to break up executable code into smaller blocks is to form it into distinct ‘procedures’ and ‘functions’ (‘sub-routines’). A ‘function’ returns something; a ‘procedure’ (what in C terms would be a ‘void function’) does not:

```
procedure Foo;
begin
  WriteLn('This is Foo speaking');
  WriteLn('It has nothing much to say really');
end;

function InchesToCentimetres(Inches: Double): Double;
begin
  Result := Inches * 2.54;
end;
```

To give a sub-routine data to work on, you define it with ‘parameters’. These are declared inside brackets immediately after the sub-routine name; in the examples just given, `Foo` is parameterless, but `InchesToCentimetres` has a single parameter called `Inches` that is typed to `Double`. The general form for parameter declarations is ParamName: TypeName.

When a routine takes more than one parameter, they are declared delimited with semi-colons, e.g. `(Param1: Real; Param2: Integer)`. When adjacent parameters have the same type, you can however use a slightly more compact style in which parameter names are delimited with commas, and the type stated only once at the end: `(Param1, Param2: Double; StrParam1, StrParam2: string)`. Either way, when the routine is *called*, parameter values — ‘arguments’ — are always separated by commas: `MyProc(Arg1, Arg2, Arg3)`.

When a sub-routine takes no parameters, you are free to use either an empty pair of brackets or no brackets at all. This holds both when the routine is declared and when it is called:

```
procedure Foo2();
begin
  WriteLn('This is Foo2 speaking');
end;

procedure Test;
begin
  Foo();
  Foo2;
end;
```

The convention is to leave them out, perhaps because traditional Pascal did not give you the option of using empty brackets in the first place, however it’s really just a matter of taste.

Parameter passing

By default, parameters are passed by ‘value’. This means the source is copied before it is presented to the procedure or function, which can then change it at will without affecting the caller:

```
procedure Foo(Value: Integer);
begin
  Value := 100;
end;

var
  Value: Integer;
begin
  Value := 20;
  Foo(Value);
  WriteLn(Value); //outputs 20
end.
```

Alternatively, `const`, `var` and `out` modifiers are available for parameters that, respectively, shouldn’t be modified within the called routine, that should be passed by reference, and that denote values to be returned (but not passed in). Or at least, that’s the basic principle — the details are a bit more complicated.

Constant parameters

While it works as expected for most types, for class and dynamic array parameters, `const` doesn’t actually prevent the source object or array from being modified:

```
uses System.SysUtils;

procedure NotSoConst(const Arr: TArray<string>;
  const Obj: TStringBuilder);
begin
  Arr[0] := 'Er, what''s up with this?';
  Obj.Append('Funny sort of "const"!');
end;
```

The reason for this behaviour is that classes and dynamic arrays are pure reference types. Because of that, using `const` only prevents you from from changing the reference, not the data it points to. You might then expect strings to exhibit similar behaviour, given they are reference types too. However, the compiler implements quasi-value type semantics for strings:

```
type
  TMyStaticArray = array[0..4] of string;

  TMyRec = record
    X, Y: Integer;
  end;

procedure MuchBetter(const Arr: TMyStaticArray;
  const R: TMyRec; const S: string; const V: Variant);
begin
  Arr[0] := 'Won''t compile';
  R.X := 100;  //won't compile
  S[1] := '!'; //won't compile
  S := 'Won''t compile either';
  V := 'Nor will this';
end;
```

While amongst the reference types `const` only makes strings actually constant, it is still a good idea to use, assuming the routine isn’t intending to change the values passed. In the case of dynamic arrays and other managed types, it relieves the compiler from having to increment the argument’s reference count on entry and decrement it on exit; and for value types whose instances are more than a few bytes, `const` will cause the compiler to pass internally just a pointer to the source data rather than a copy of it, which is more efficient.

Variable parameters

To be a ‘variable’ parameter means that its value can be changed inside the routine, and moreover, that any change is performed on the original value:

```
procedure Foo(var Value: Integer);
begin
  Value := 100;
end;

procedure TestFoo;
var
  Value: Integer;
begin
  Value := 20;
  Foo(Value);
  WriteLn(Value); //outputs 100
end;
```

The type compatibility of variable parameters is stricter than that enforced by value and constant ones. For example, if in `TestFoo` above `Value` were typed to `Byte` rather than `Integer`, the snippet would no longer compile; remove the `var` from the parameter declaration, however, and it would.

Out parameters

In Delphi, `out` parameters behave exactly the same as `var` ones except in the case of managed types. There, using `out` rather than `var` causes inputted values to be automatically cleared on entry:

```
procedure TestChar(out Ch: Char);
begin
  WriteLn(Ch);   //Ch has a non-managed type, so won't be cleared
end;

procedure TestString(out AStr: string);
begin
  WriteLn(AStr); //AStr has a managed type, so will be cleared
end;

var
  Ch: Char;
  Str: string;
begin
  Ch := 'A';
  TestChar(Ch);     //outputs A
  Str := 'testing';
  TestString(Str);  //outputs nothing
end.
```

Default parameter values (optional parameters)

With a few limitations, both value and constant parameters can be declared to have a default value, making their explicit passing optional:

```
procedure Foo(const Name: string = 'John Smith';
  Age: Integer = 55);
begin
  WriteLn(Name, ' is ', Age);
end;

begin
  Foo;                 //output: John Smith is 55
  Foo('Joe Bloggs');   //output: Joe Bloggs is 55
end.
```

When being declared, all parameters to the right of the first parameter given a default value must be given one too. In the above case, that means since `Name` has a default value, `Age` must have one too. A further limitation is that default values themselves must either be literals or true constants, i.e. something whose value is determined at compile time. As a result, the only valid default value for most reference and pointer types (`string` being the exception) is `nil`. If you desire anything else, you must overload the routine instead (see below).

Function return values

The return value of a function must always have a type, which like a parameter's is specified with a colon followed by the type name. If a function has parameters, then its return type always follows *after* the parameter list.

Within the body of a function, actually set the return value by assigning the `Result` pseudo-variable. Until the function exits, this may be set (and indeed read) any number of times:

```
function DitheringFunc: Integer;
begin
  Result := 3;
  Result := Result * 11;
  Result := Result + 15;
  Result := Result - 6;
end;
```

On entry, the initial value of `Result` should be considered undefined. The compiler will therefore warn you if you allow for a code path that doesn't assign `Result`:

```
function IsTheAnswer(Value: Integer): Boolean;
begin
  if Value = 42 then
    Result := True;
end;

[DCC Warning] Project1.dpr(12): W1035 Return value of function 'IsTheAnswer' might be undefined
```

When called, it is perfectly legal to pretend a function is just a procedure:

```
var
  FoundIt: Boolean;
begin
  FoundIt := IsTheAnswer(99); //OK
  IsTheAnswer(42);            //OK too
```

If called as a procedure, like the second call to `IsTheAnswer` here, the return value just gets thrown away.

Prematurely exiting from a sub-routine

To immediately exit from a sub-routine, call `Exit`:

```
procedure Foo(const Str: string);
begin
  if Str = '' then Exit;
  WriteLn(Str);
end;
```

In the case of a function, `Exit` can optionally set `Result` at the same time by passing the desired value to it:

```
function IsOK(const Str: string): Boolean;
begin
  if Str = '' then Exit(False);
  //other stuff...
end;
```

Sometimes taking the effort to set `Result` explicitly leads to more readable code however.

Overloading

Routines can be 'overloaded', meaning that you can have two or more routines with the same name yet with different parameter lists. To enable this feature, each version of the routine must be declared with the `overload` directive:

```
procedure SayHello(const FullName: string); overload;
begin
  WriteLn(FullName, ' says hello');
end;

procedure SayHello(const FirstName, Surname: string); overload;
begin
  WriteLn(FirstName + ' ' + Surname, ' says hello');
end;

procedure SayHello(Number: Integer); overload;
begin
  WriteLn('The number ', Number, ' says hello');
end;
```

To be valid, the compiler must be able to disambiguate between different overloads. Nonetheless, it can disambiguate pretty well; for example, adding the following overload to the three just presented is formally legal, if poor style:

```
procedure SayHello(Number: Byte); overload;
begin
  WriteLn('The byte-sized number ', Number, ' says hello');
end;

var
  Value: Integer;
begin
  Value := 42;
  SayHello(Value); //calls the Integer version
  SayHello(42);    //calls the Byte version
end.
```

However, the following won't compile:

```
function SayHello(const FullName: string): Boolean; overload;
begin
  Result := (FullName <> '');
  if Result then WriteLn(FullName, ' says hello');
end;
```

The problem here follows from the fact a function can be called as if it were a procedure: if this final variant were called that way, it would be impossible for the compiler to disambiguate from a call to the first version of `SayHello`.

Nested routines

Inside any procedure or function can lie 'nested' routines, which may be declared anywhere in between the outer routine's header and its `begin` keyword. These nested routines can only be accessed by their outer routine, but have themselves access to both the outer routine's own arguments and any local variables that are declared prior to the nested routine itself.

Nested routines are particularly useful in recursive situations, i.e. when a routine may call itself. Here, you code the nested routine as the recursive one, and leave the outer routine to perform any preliminary work before calling the nested routine. In the following example, the pattern is followed by a procedure that recursively assigns the system font to a VCL form and its controls:

```
uses System.SysUtils, Vcl.Controls, Vcl.Forms;

type
  TControlAccess = class(TControl);

{ Apply system font settings, taking account of how the
```

```
  ParentFont property of a child control will be False
  even if just its font size differs from the parent's. }

procedure ApplySystemFont(Form: TForm);
var
  OldName, NewName: string; //declared above the nested routine

  procedure CycleChildren(Parent: TWinControl);
  var
    Child: TControl;
    I: Integer;
  begin
    for I := 0 to Parent.ControlCount - 1 do
    begin
      Child := TControlAccess(Parent.Controls[I]);
      if not TControlAccess(Child).ParentFont and
         SameText(OldName, TControlAccess(Child).Font.Name) then
      begin
        TControlAccess(Child).Font.Name := NewName;
        if Child is TWinControl then
          CycleChildren(TWinControl(Child));
      end;
    end;
  end;
begin
  OldName := Form.Font.Name;
  Form.Font := Screen.MessageFont;
  NewName := Font.Name;
  CycleChildren(Form);
end;
```

Procedural pointers

Delphi allows you to define C-style procedural pointer types, sometimes called ‘callback’ types. This relatively advanced feature allows you to declare a variable or parameter that gets assigned a standalone routine, or more accurately, a reference to a procedure or function of a given signature. The assigned routine can then be invoked through the variable or parameter as if it were the routine itself. This ‘callback’ style of programming is used quite a lot by the Windows API. For example, to enumerate all top level windows currently running on the system, you can call the `EnumWindows` API function; as used, this takes a reference to a function of your own that then gets called repeatedly for each window found.

The syntax for defining a procedural pointer type takes the form of an identifier being made to equal a nameless procedure or function. In the following example, a type is declared for a function that takes and returns a string:

```
type
  TTransformStringFunc = function (const S: string): string;
```

Once the type is defined, you can declare instances of it, which can then be assigned routines of the specified signature. ‘Signature’ here means matching parameter and return types — names are irrelevant:

```
uses
  System.SysUtils, System.Character;

type
  TTransformStringFunc = function (const S: string): string;

function TitleCase(const Source: string): string;
var
  I: Integer;
  InSpace: Boolean;
begin
  Result := ToLower(Source);
  InSpace := True;
  for I := 1 to Length(Result) do
    if IsWhiteSpace(Result[I]) then
      InSpace := True
    else
    begin
      if InSpace and IsLetter(Result[I]) then
        Result[I] := ToUpper(Result[I]);
      InSpace := False;
    end;
end;

var
```

```
  Func: TTransformStringFunc;
begin
  { assign our custom function }
  Func := TitleCase;
  { invoke it }
  WriteLn(Func('TEST STRING')); //output: Test String
  { assign a function from the RTL with a matching signature }
  Func := ToLower;
  { invoke it }
  WriteLn(Func('TEST STRING')); //output: test string
end.
```

Here, the `Func` variable is first assigned a reference to our custom `TitleCase` routine, invoked, then secondly assigned `ToLower` and invoked once more. This has the same effect is as if `TitleCase` and `ToLower` had been called directly.

A procedural pointer of any type also accepts the value `nil`, meaning 'unassigned'. To test for `nil`, you can either 'reference' the variable with the @ sign or call the `Assigned` standard function:

```
procedure Test(Item1, Item2: Pointer;
  const Callback: TSimpleCallback);
var
  RetVal: Boolean;
begin
  if Assigned(Callback) then
    RetVal := Callback(Item1, Item2);
  if @Callback <> nil then
    RetVal := Callback(Item1, Item2);
end;
```

Type safety and procedural pointers

By default, it is impossible to assign a routine of the wrong signature to a procedural pointer. However, normal type safety can be overridden by explicitly referencing the assignee with an @:

```
type
  TSimpleCallback = function (Item1, Item2: Pointer): Boolean;

function CallbackForObjs(Item1, Item2: TObject): Boolean;
begin
  if Item1 = nil then
    Result := Item2 = nil
  else
    Result := Item1.Equals(Item2);
end;

procedure Test;
var
  Callback: TSimpleCallback;
begin
  Callback := @CallbackForObjs; //force assignment
end;
```

Using @ like this is a bad habit though — in this case it works, since a `TObject` has the same size as a `Pointer`, however if you get it wrong, nasty crashes may well result at runtime.

Unfortunately, it will nevertheless be necessary to use @ if you need to call a routine that takes a callback, and the callback parameter has been 'weakly' typed to `Pointer`. Annoyingly enough, many of the Windows API functions that take callbacks, like the aforesaid `EnumWindows`, are declared like that. This means you must be extra careful when using them (check out chapter 13 for an example).

Units

So far, we have been writing code in the context of a single source file, the DPR. In practice, the substance of an application is typically elsewhere though, spread out into separate 'units' instead.

For example, if you use the IDE to create a new VCL Forms application, the DPR itself will look like this:

```
program Project1;

uses
  Vcl.Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.res}

begin
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

The main bulk of even the empty project will be in a separate unit, here `Unit1`:

```
unit Unit1;

interface

uses
  Winapi.Windows, Winapi.Messages, System.SysUtils,
  System.Variants, System.Classes, Vcl.Graphics, Vcl.Controls,
  Vcl.Forms, Vcl.Dialogs;

type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

end.
```

Each time you add a new form to the application, a new unit will be created. Further, additional classes and routines will normally be defined in their own units too, rather than cluttering up the DPR.

Interface vs. implementation sections

A unit's source code is written to a file with a PAS extension, with one PAS file per unit. Similar to a DPR, a PAS file starts with the keyword `unit` (in place of `program`) followed by its identifier, and finishes with the word `end` and a full stop (period if you speak American).

As with a program, a unit must be named with a valid Delphi identifier, albeit with one small restriction lifted: a unit name, unlike any other identifier, can include one or more full stops. Starting with XE2, most standard units make use of this in order to indicate their place in the wider scheme of things. Thus, the names of basic units are prefixed with `System.` (e.g. `System.StrUtils`), database-related units by `Data.`, VCL units prefixed by `Vcl.`, and so on. However, you don't have to use so-called 'unit scope' names yourself — calling a unit simply `MyUtils` or whatever is perfectly OK. In fact, for reasons of backwards compatibility, you don't even have to prefix the standard units with their unit scope name — e.g., `System.StrUtils` can be written as simply `StrUtils`. For compiler performance with larger projects it is a good idea to use the full names of standard units though.

Unlike a DPR, a PAS file is structured into two parts, an 'interface' section and an 'implementation' section. Immediately prior to its closing `end.`, a unit may also have `initialization` and `finalization` blocks. If a `finalization` block is defined, an `initialization` one must be too, if only to be left empty:

```
unit MyUnit;
```

```
interface

uses
  ImportedUnit1, ImportedUnit2;

{ Exported types, variables and constants go here... }

implementation

uses ImportedUnit3, ImportedUnit4;

{ Any private types, variables and constants go here, together
  with the implementation of public and private types alike... }

initialization

{ Statement(s)... }

finalization

{ Statement(s)... }

end.
```

The code in a unit's initialization section will be executed on program startup if linked into an EXE, or when the shared library is loaded if compiled into a DLL (Windows) or dylib (OS X). Any finalization section is called on program or library shutdown.

Declaring unit members

As with a DPR, the types, variables and constants of a unit are declared in `type`, `var` and `const` blocks respectively. Any number of these blocks can appear in any order, and multiple items can appear in each:

```
type
  TMyObject = class
    Str: string;
  end;

const
  MeaningOfLife = 42;
  TypedConst: Single = 1.23456;

var
  MyGlobal: Integer; AnotherGlobal: string;

type
  TMyRec = record
    Int: Integer;
  end;
```

These blocks may appear in the interface section, the implementation section, or both.

The uses clause

Both `interface` and `implementation` sections can also have their own `uses` clause, and once again, both are optional. What a `uses` clause does is to import (i.e., make accessible) the exported symbols (i.e., the sub-routines, types, variables, constants, etc.) of other units. The following uses clause, therefore, imports the public symbols of the `System.SysUtils`, `System.Classes` and `MyUtils` units:

```
uses
  System.SysUtils, System.Classes, MyUtils;
```

A `uses` clause in Delphi is not merely a shortcutting mechanism, as (for example) a `using` statement in C# is: if a unit does not appear, then its symbols will be inaccessible. The only exceptions to this are the built-in `System` and `SysInit` units, which are implicitly always imported.

The order in which units are listed in a uses clause can be important: if the same identifier is exported by more than one imported unit, the unit that comes *later* in the uses clause gets priority. For example, say `MyUnit` defines a string constant called `MyConst` that is set to `'Hello'`, and `MyOtherUnit` also defines a string constant called `MyConst`, only this time set to `'Goodbye'`. In the following case, `Goodbye` will be outputted:

```
uses
  MyUnit, MyOtherUnit;
```

```
begin
  WriteLn(MyConst);
end.
```

If that is not the one you want, then you must either rearrange the uses clause, or alternatively, ‘fully qualify’ the symbol with its unit name and a dot:

```
uses
  MyUnit, MyOtherUnit;

begin
  WriteLn(MyUnit.MyConst);
end.
```

In this case, `Hello` will be outputted.

Exporting symbols

To export a symbol yourself, you declare it in your unit’s `interface` section. In the case of a variable or constant, this means the whole declaration goes in that section; in the case of a class or record, just the type definition goes there, any method bodies being left to the `implementation` section:

```
unit UnitFoo;

interface

type
  TFoo = class
    procedure SayHello;
  end;

implementation

procedure TFoo.SayHello;
begin
  WriteLn('Hello!');
end;

end.
```

Forward declarations

In general, a type has to be declared before it can be used. ‘Forward declarations’ are nevertheless possible in the case of both classes and standalone routines. Only in the case of the latter is the `forward` keyword actually used though:

```
implementation

type
  TFoo = class;             //forward declaration of class

  TFooEvent = procedure (Sender: TFoo) of object; //use it

  TFoo = class
  strict private
    FOnGoodbye: TFooEvent;
  public
    procedure SayGoodbye;
    property OnGoodbye: TFooEvent read FOnGoodbye write FOnGoodbye;
  end;

procedure Bar; forward; //forward declaration of standalone routine

procedure TFoo.SayGoodbye;
begin
  Bar;                  //make use of it
  if Assigned(FOnGoodbye) then FOnGoodbye(Self);
end;

procedure Bar;
begin
  WriteLn('Goodbye');
end;
```

Forward class declarations have one important restriction, namely the fact they are only valid so long as the class comes to be fully defined in the same `type` block. The following code, therefore, will not compile:

```
type
  TFoo = class;

const
  MeaningOfLife = 42;

type
  TFoo = class
    procedure SayHello;
  end;
```

A bit inconsistently, record types cannot have forward declarations at all. The compiler nonetheless allows you to define a typed pointer type to a record before you declare the record itself. The following is therefore valid:

```
type
  PMyRec = ^TMyRec;

  TNotInTheWayObject = class
  end;

  TMyRec = record
    Int: Integer;
  end;
```

As with forward class declarations, a 'same `type` block' rule holds here too. Moreover, the convention is to define a record's typed pointer type immediately before the record itself, so unless you have good reason not to, it's best to do that.

Unit file lookup

In the terminology of the IDE, a unit can be 'added' to a project. When this happens, the unit in question is added to the DPR's uses clause in a special format that includes the path of its source file, relative to the DPR itself:

```
uses
  System.SysUtils,
  Unit1 in 'Unit1.pas',         //in same directory as DPR
  Unit2 in '..\Unit2.pas',      //in the parent directory
  Unit3 in '\SubDir\Unit3.pas'; //in sub-directory called SubDir
```

If a unit is not explicitly added to a project like this, then as and when it is used by a unit that is, the compiler uses a directory-based lookup. Here, the DPR's directory is looked in first. In the IDE, further directories may be specified either via the global 'library path' setting (`Tools|Options`, `Environment Options>Delphi Options>Library`) or the project-specific 'search path' (`Project|Options`, `Delphi Compiler`). Both will be pre-filled with several standard directories that contain the DCU (= compiled unit) files for Delphi's standard libraries — don't remove them, else you will have difficulty compiling all but the very simplest projects.

Source files and compiled units

When a unit source file is compiled, a DCU (Delphi Compiled Unit) file is produced. This then means the compiler can avoid having to recompile a PAS file whenever you compile a program that uses it — if nothing has changed in the unit itself, then the compiler will happily reuse the old DCU. In fact, in principle, the compiler doesn't even have to have access to the PAS file: if it can't find the PAS but can find a corresponding DCU, it will link in that DCU without complaint, assuming it is compatible.

As for what 'compatible' means, DCU files are unfortunately highly sensitive to compiler versions. The DCU files produced by Delphi XE and Delphi XE2, for example, are mutually incompatible, though the DCU files of XE2 Starter will work with XE2 Architect and vice versa. In other words, different *editions* of Delphi share the same compiler (and therefore, produce compatible DCUs) so long as they are of the same *version*.

Whether a DCU needs to be recompiled or not is taken care of by the compiler automatically. Attempting to use a unit that only has a DCU created by an earlier (or later) version of Delphi will therefore cause the compiler to regenerate the DCU; and if the source PAS file cannot be found, a normal compiler error will result. The same thing happens on symbols nominally imported from a DCU not corresponding with the symbols exported by it, or (in the case of its PAS file being findable by the compiler) when it is out of date relative to its PAS source.

Error handling

Delphi uses an exception model of error handling, in which you write code in a way that generally *ignores* the possibility of errors. Thus, instead of errors being indicated in the form of error codes you must explicitly check for, they get 'raised' as 'exceptions' that forcibly interrupt the current line of execution unless explicitly handled.

If you aren't used to the exception model, it can sound scary at first, but it isn't really. The reason is that an unhandled exception percolates up the call stack: if `FuncA` calls `FuncB`, which itself calls `FuncC` which then raises an exception, a failure of `FuncB` to handle the error just means `FuncA` now has the opportunity to do so instead; and if that doesn't either, then what called `FuncA` now has and so on.

In the case of a GUI application, this leads to the concept of a 'default exception handler': if deep into the call stack an exception gets raised but never handled, the stack progressively unwinds until the GUI framework handles the exception itself, showing a nicely-formatted error message to the user. This all contrasts to an error code model, in which the code closest to an error must trap it else it won't get trapped at all, greatly increasing the risk of data corruption and worse.

Handling errors: try/except blocks

The actual mechanics of handling an exception takes the form of a try/except block:

```
try
  //do stuff that may raise an exception
except
  //handle any exception
end;
```

Here, any exception raised after the `try` but before the `except` will see execution move to the `except` block. Writing a general exception handler like this is bad style however. Instead, you should be checking for specific exception types — since exceptions in Delphi are represented by objects, this means checking for specific class types:

```
try
  //do stuff that may raise an exception
except
  on Ident: ExceptionClass1 do
  begin
    { handle first sort of exception... };
  end;
  on Ident: ExceptionClass2 do
  begin
    { handle the second };
  end;
end;
```

If the actual exception type is not matched, the exception is not caught and the `except` block effectively ignored.

Any number of `on` clauses can lie inside an except block. With each, specifying an identifier is optional; if you merely want to check for the exception type, you can use a slightly abbreviated syntax:

```
on ExceptionClass do Statement;
```

Otherwise, the additional identifier declares an inline local variable which gets automatically assigned the raised exception object on being matched. Since each `on` statement is independent from the others, you can reuse the same identifier in a way you couldn't with an ordinary local variable.

Under the hood, matching is performed using a series of nested `if` statements that use `is` tests. Because of this, you need to be a bit careful about how you order your `on` statements. Consider the following code:

```
uses System.SysUtils;

var
  IntValue: Integer;
begin
  IntValue := 0;
  try
    IntValue := 42 div IntValue;
  except
    on E: Exception do
      WriteLn('Error: ', E.Message);
    on EDivByZero do
      WriteLn('Integral division by zero error!');
    else
      raise;
  end;
```

```
end.
```

Since `Exception` is the base exception class type (more on which shortly), `EDivByZero` descends from it. As `Exception` is checked for first however, the second `on` statement will never be called.

Raising exceptions

Inside of an exception handler, you can re-raise the original exception via the `raise` statement:

```
except
  on EDivByZero do
  begin
    WriteLn('Integral division by zero error!');
    raise; //re-raise the original exception
  end;
end;
```

On the exception being re-raised, the flow of execution will move up the call stack as if the exception hadn't been handled in the first place.

How to raise a new exception rather than re-raise an old one is not radically different — once again, the `raise` statement is used, only this time with a constructed object:

```
procedure Foo(Obj: TObject);
begin
  if Obj = nil then
    raise EArgumentNilException.Create('Obj cannot be nil!');
  WriteLn(Obj.ClassName);
end;
```

Once an object is passed to `raise`, the RTL now 'owns' it, and will free it automatically.

Formally, any object whatsoever can be raised as an exception object. However, the `System.SysUtils` unit defines the `Exception` type to serve as a base class, and all standard exception types descend from that. In itself, the main thing `Exception` provides from a user perspective is a `Message` property, set via the constructor. To the constructor can be passed either a string expression like the example just given, or a pointer to a 'resource string':

```
resourcestring
  SStringBuildersNotAllowed = 'String builders not allowed!';

procedure Foo2(Obj: TObject);
begin
  if Obj is TStringBuilder then
    raise EArgumentException.CreateRes(@SStringBuildersNotAllowed);
  WriteLn(Obj.ClassName);
end;
```

While you could just pass a string resource directly — `EArgumentException.Create(SStringBuildersNotAllowed)` — the pointer-y version means the resource only gets loaded if it is actually used.

Opinions can differ, but in my view standard exception types declared in units such as `System.SysUtils` and `System.Classes` should be used as much as possible. This is because there is no way of telling what exceptions a Delphi routine might raise short of reading the source code, both of the routine itself *and* anything it calls (remember, unhandled exceptions percolate up the call stack). Consequently, the fact your own code will only explicitly raise `EMyProductError`-derived exceptions or whatever is not to say it will *only* raise them. From the calling code's point of view however, the less possible exception types it might wish to trap the better.

Another thing to keep in mind is that raising exceptions is typically a library-level rather than application-level thing to do. While you could use exceptions as a sort of generic signalling mechanism, in which an exception being raised doesn't necessarily mean something bad has happened, that isn't what they are designed for.

Ensuring clean up code runs: try/finally blocks

As such, `try`/`except` blocks are not things that should be liberally littered through your code. If exceptions are being raised as a matter of course, then you should be coding to avoid them being raised in the first place; and if they aren't, adding try/except blocks 'just the case' would be to rather miss the point of exceptions.

However, there is another sort of `try` statement that *should* be used on a frequent basis: `try`/`finally` statements. These are to guarantee that specified 'clean up' code will be run. Their general form is the following:

```
try
  Statement(s)
finally
```

```
  Statement(s)
end;
```

If an exception is raised between the `try` and the `finally`, the line of execution moves to the finally block; and if no exception happens, the line of execution moves through it as if the `try`/`finally`/`end` keywords were never there. Whether a `try`/`except` block is in between the `try` and the `finally` makes no difference to this.

Alternatively, if it is the `try`/`finally` that is encased inside a `try`/`except` block, the `finally` part is executed prior to the `except` part:

```
uses
  System.SysUtils;

begin
  try
    try
      raise Exception.Create('Test exception');
    finally
      WriteLn('Finally');
    end;
  except
    Writeln('Except');
  end;
end.
```

The above code will therefore output `Finally` followed by `Except`.

Pointers and dynamically allocated memory

For most types, memory is allocated and deallocated for you, either entirely automatically (value types, strings), or by following a specific pattern (the `Create` and `Free` methods of a class). In neither case is there any ambiguity about whether memory is allocated correctly, since if you don't do it the proper way, your code will not even compile.

Nonetheless, it is also possible to allocate memory more or less directly. Doing so requires using pointers, and between them they form a distinctly low-level and error prone way of programming. Because of this, it isn't usual for Delphi code to make frequent or even occasional recourse to them. Put another way, if you find the next few pages a lot more hard-going than what came before, that's to be expected! Given pointers are a tool Delphi code 'in the wild' may use though, it is good to at least be aware of them.

Allocating memory dynamically

To explicitly allocate a block of memory, call either `GetMem`, `AllocMem` or `New`:

```
var
  P1, P2: PByte;
  P3: PInteger;
begin
  GetMem(P1, 10);    //allocated 10 bytes of memory
  P2 := AllocMem(20) //allocate 20 more bytes
  New(P3);           //allocate SizeOf(Integer) bytes of memory
```

`GetMem` is the lowest level routine, and simply allocated the specified number of bytes. `AllocMem` then does the same, but zero-initialises the bytes before returning:

```
var
  DblPtr1, DblPtr2: PDouble;
begin
  GetMem(DblPtr1, SizeOf(Double));
  DblPtr2 := AllocMem(SizeOf(Double));
  WriteLn(DblPtr1^);            //output: who knows?
  WriteLn(DblPtr2^);            //output: 0.00000000000000E+0000
```

Lastly, `New` works with typed pointers, and allocates the specified number of *elements* of the pointed-to type — `GetMem` and `AllocMem`, in contrast, allocate a specified number of *bytes*. When the pointed-to type is a managed one such a string, or a record that contains a string, `New` also zero-initialises the memory. This is because managed types require that to happen if they are to work correctly.

When you have finished with a dynamically-allocated memory block, call `FreeMem` to dispose of it if `GetMem` or `AllocMem` had been called, or `Dispose` if `New` had been:

```
var
  PI: PInteger;
  PV: PVariant;
begin
  GetMem(PI, SizeOf(Integer));
  try
    //...
  finally
    FreeMem(PI);
  end;
  New(PV);
  try
    //...
  finally
    Dispose(PV);
  end;
```

If using `FreeMem` rather than `Dispose` and the memory includes an instance of a managed type, call `Finalize` immediately beforehand so that it is disposed of properly.

In practice, there is rarely a good reason to use Delphi's low-level memory management routines. In particular, if you want to allocate an arbitrarily-sized block of bytes, `TBytes` — a dynamic array of `Byte` — forms a much more convenient and safer alternative. In the `TBytes` case, you allocate memory by calling `SetLength`; once the variable goes out of scope, the memory will then be deallocated automatically — there is no need to call `FreeMem` or `Dispose`:

```
var
  ManualBytes: PByte;
  EasyBytes: TBytes;
begin
  //the GetMem/FreeMem way
  GetMem(ManualBytes, 128);
  try
    ManualBytes[0] := 1;
    ManualBytes[1] := 2;
  finally
    FreeMem(PI);
  end;
  //the TBytes way - no explicit deallocation needed
  SetLength(EasyBytes, 128);
  EasyBytes[0] := 1;
  EasyBytes[1] := 2;
```

If you want to deallocate the memory pointed to by a `TBytes` instance immediately, just set it to `nil`:

```
  EasyBytes := nil;
```

Working with pointers

However the memory they point to is allocated, pointers are used the same way: get a pointer to something using the `@` sign, and 'dereference' the pointer — retrieve the value pointed too — using `^` symbol. Further, all pointers can be tested for equality, in which case the addresses they point to are compared:

```
var
  I1, I2: Integer;
  P1, P2: Pointer;
begin
  I1 := 22;
  I2 := 22;
  P1 := @I1;
  P2 := @I2;
  WriteLn(P1 = P2); //output: FALSE
  P2 := @I1;
  WriteLn(P1 = P2); //output: TRUE
```

To perform a binary comparison of the bytes pointed to instead, call the `CompareMem` function, as declared in `System.SysUtils`:

```
uses
  System.SysUtils;

var
  Ints1, Ints2: array[0..1] of Integer;
  P1, P2: PInteger;
begin
  Ints1[0] := 123;
  Ints1[1] := 456;
  //assign same data to Ints2...
  Ints2[0] := 123;
  Ints2[1] := 456;
  //assign pointers to point to start of respective array
  P1 := @Ints1[0];
  P2 := @Ints2[0];
  //same addresses?
  WriteLn(P1 = P2);                           //output: FALSE
  //same data pointed to?
  WriteLn(CompareMem(P1, P2, SizeOf(Ints1))); //output: TRUE
```

Typed pointers specifically also support the `Inc` and `Dec` standard routines. These are usually used with ordinal types, in which case the value is incremented or decremented by one or (if a second parameter is passed) the number specified. Thus, if `I` and `J` are both integers holding the value 4, then calling `Inc(I)` changes I to 5, and calling `Dec(J, 2)` sets J to 2.

Since the value of a pointer is a memory address, calling `Inc` or `Dec` changes the address:

```
var
  Bytes: TBytes;
  Ptr: PByte;
begin
  SetLength(Bytes, 2);
  Bytes[0] := 111;
  Bytes[1] := 222;
  Ptr := @Bytes[0]; //reference the first element
  WriteLn(Ptr);     //output: 111
  Inc(Ptr);         //move the pointer up one place
  WriteLn(Ptr);     //output: 222
```

To increment or decrement the thing pointed to instead, the pointer must be dereferenced first:

```
var
  Int: Integer;
  Ptr: PInteger;
begin
  Int := 100;
  Ptr := @Int;
  Inc(Ptr^, 50); //increment the value pointed to by 50
  WriteLn(Ptr^); //150
```

A subtlety of `Inc` and `Dec` when used with pointers is that they increment and decrement in terms of the thing pointed to. When that thing is only one byte in size, as in the `PByte` example just now, that does not matter. It does matter otherwise however:

```
var
  Ints: TArray<Int16>; //dynamic array of 16bit integers
  Ptr: PInt16;
begin
  Ints := TArray<Int16>.Create(111, 222, 333);
  Ptr := @Ints[0];
  Inc(Ptr, 2);   //move address on by by 2 elements, not 2 bytes
  WriteLn(Ptr^); //output: 333
```

If `Inc` incremented in terms of bytes not elements, `Ptr` would now point to the second not the third element of the array, and therefore, point to the value 222, not 333. This can be seen if `PByte` rather than `PInt16` is used for the pointer type:

```
var
  Ints: TArray<Int16>; //dynamic array as before
  Ptr: PByte;
begin
  Ints := TArray<Int16>.Create(111, 222, 333);
  Ptr := @Ints[0]; //reference the first element
  Inc(Ptr, 2);     //move the address on by two bytes
  WriteLn(Ptr^);   //output: 222
```

As shown here, it is the type of the pointer that determines how many bytes `Inc` and `Dec` assume one element takes up, not what is 'actually' pointed too.

A related trick supported by certain sorts of pointer by default is 'pointer arithmetic'. This in effect generalises the support for `Inc` and `Dec` into support for the +, -, <, <=, > and >= operators too:

```
var
  SourceStr: string;
  P1, P2: PChar;
begin
  SourceStr := 'Pointer maths is... interesting';
  P1 := PChar(SourceStr);
  { Point P2 to the final character }
  P2 := P1 + Length(SourceStr) - 1;
  WriteLn(P2^);      //output: g
  { Does P2 point to a place later than P1? }
  WriteLn(P2 > P1);  //output: TRUE
  { Return number of characters between P2 and P1 }
  WriteLn(P2 - P1);  //output: 30
```

When one pointer is subtracted from another, the number of elements in between is returned. The compiler will happily allow you to do this even if the two pointers point to unrelated places in memory; while comparing two such pointers is not an error, you won't get anything meaningful returned either.

By default, pointer arithmetic is only supported by the `PByte`, `PChar`/`PWideChar` and `PAnsiChar` types. It can however be enabled for other pointer types via the `$POINTERMATH` compiler directive. This has two modes. When enabled prior to a pointer type being declared, the type will always support the feature:

```
type
  TMyRec = record
    Value: Integer;
  end;
{$POINTERMATH ON}   //Enable for the type(s) that follow
  PMyRec = ^TMyRec; //Define the PMyRec type
{$POINTERMATH OFF}  //Disable pointer arithmetic again

var
  Recs: array[0..1] of TMyRec;
  P1, P2: PMyRec;
begin
  P1 := @Recs[0];
  P2 := P1 + 1;
```

Alternatively, pointer arithmetic can be enabled for a specific block of code:

```
var
  Ints: array[0..1] of Integer;
  P1, P2: PInteger;
begin
  Ints[0] := 80;
  Ints[1] := 90;
  P1 := @Int;
  {$POINTERMATH ON}  //Enable for the code that follows
  P2 := P1 + 1;
  {$POINTERMATH OFF} //Disable pointer arithmetic again
```

The effect of the directive is ‘local’ in the sense it will only affect the code in the unit it appears, and even then from where it is placed downwards.

In the examples just given it is explicitly disabled soon afterwards. However, it is perfectly acceptable not to bother doing that.

2. Simple types: enumerations, numbers and date/times

The previous chapter gave a broad overview of Delphi's standard types. In this and the next few chapters, individual types will be looked at in much more detail, with relevant parts of Delphi's wider runtime library introduced in step.

Enumerated types

An enumerated type is an ordered set of identifiers:

```
type
  TDayOfWeek = (Mon, Tues, Weds, Thurs, Fri, Sat, Sun);
```

When a variable is typed to `TDayOfWeek`, it will be assignable to one of `Mon`, `Tues` etc. The fact enumerations are 'ordered' means the value `Tues` is considered greater than `Mon`, but less than `Weds` which is itself less than `Thurs` and so on. The following will therefore output `Fri is after Weds`:

```
var
  Day: TDayOfWeek;
begin
  Day := Fri;
  if Day < Tues then
    WriteLn('Fri is before Tues... huh?')
  else if Day > Weds then
    WriteLn('Fri is after Weds');
```

Unless you use a 'cast' to override normal typing rules, the only thing assignable to an enumerated type is another instance of it. In particular, it is illegal to assign an integer to an enumerated value:

```
var
  Value: TDayOfWeek;
begin
  Value := 1; //won't compile
```

Correspondingly, an enumerated value cannot be directly assigned to an integer:

```
var
  DayValue: TDayOfWeek;
  IntValue: Integer;
begin
  DayValue := Fri;
  IntValue := DayValue; //won't compile
```

Ordinal type standard routines

As with all ordinal types, enumerations support a small group of intrinsic routines ('intrinsic' in the sense they are built into the language):

- `Ord` returns the underlying integral value of the element. Unless explicitly overridden, the ordinal value of the first element of an enumeration is 0, the second 1 and so on:

```
var
  Day: TDayOfWeek;
begin
  WriteLn(Ord(Mon));   //output: 0
  Day := Sat;
  WriteLn(Ord(Day));   //output: 6
```

- `Inc` increments the given value by one, or if a second parameter is specified, the number of values requested:

```
var
  Day: TDayOfWeek;
begin
  Day := Weds;
  Inc(Day, 3);   //Day now becomes Sat
```

  For a call to `Inc` to compile, the value passed to it must either a variable or a parameter — it cannot be a property of an object.

- `Dec` decrements the value by one, or if a second parameter is specified, the number of values requested:

```
var
  Day: TDayOfWeek;
begin
  Day := Weds;
  Dec(Day, 2);   //Day now becomes Mon
```

  Like `Inc`, `Dec` doesn't work with properties.

- `Succ` returns the successor value of the argument: `Succ(Mon)` therefore returns `Tues`.
- `Pred` returns the predecessor value of the argument: `Pred(Fri)` therefore returns `Thurs`.
- `Low` returns an ordinal type's low bound value, and `High` its highest value. Both can be passed either an instance of the

type or the type identifier directly:

```
  var
    Day: TDayOfWeek;
  begin
    Day := Low(TDayOfWeek); //assigns Mon
    Day := High(Day);       //assigns Sun
```

Enumerated type scoping

By default, enumerated type elements are 'scoped' to the unit rather than the type. This has the implication that if a given identifier is used in one enumeration, it cannot be used in another if both are in the same unit:

```
unit MyTypes;

interface

type
  TBrowser = (Chrome, FireFox, IE, Opera);
  TPerformingArt = (Dance, Opera, Theatre) //!!!compiler error
```

An attempt to compile this example will fail due to `Opera` appearing twice.

Partly because if this limitation, there is a convention to prefix the element names of an enumeration with a short prefix, typically a couple of letters. For example, the `TFontStyle` enumeration in the VCL uses an `fs` prefix:

```
type
  TFontStyle = (fsBold, fsItalic, fsUnderline, fsStrikeOut);
```

Occasionally, this may still not prevent name clashes arising when two units are used that both define a symbol with the same name. When that happens, it can be worked around by 'fully qualifying' the problematic identifier with its unit name and a dot:

```
var
  Style: TFontStyle;
begin
  Style := Vcl.Graphics.fsBold;
```

Another possibility is to use the `$SCOPEDENUMS` compiler directive. This has a couple of effects: firstly, it causes enumerated type elements to be scoped to the type itself rather than the unit, and secondly, it forces qualification to the type:

```
{$SCOPEDENUMS ON}
type
  TStarRating = (None, One, Two, Three, Four, Five);
  TDiscountLevel = (None, NormalBulk, Special);
var
  Rating: TStarRating;
  Level: TDiscountLevel;
begin
  Rating := TStarRating.None; //OK
  Level := None;              //compiler error
end.
```

In this example, using `$SCOPEDENUMS` allows defining `None` as an element of both `TStarRating` and `TResponsibilityLevel`, but at the cost of greater verbosity when the type is actually used.

Since `{$SCOPEDENUMS ON}` is a relatively new addition to the language, most enumerations defined by the RTL and other standard libraries don't use it. It is creeping into newer frameworks like FireMonkey though, and when used in them, has exactly the same effect as when you use it for your own types:

```
//using Vcl.Dialogs
MessageDlg('I think this looks cleaner', mtInformation, [mbOK], 0);
//using FMX.Dialogs
MessageDlg('Bah humbug!', TMsgDlgType.mtInformation,
  [TMsgDlgBtn.mbOK], 0);
```

Converting to/from raw ordinal values

In principle, you shouldn't care what integral value an enumerated type element has internally — all that's important is that the second element declared has an ordinal value higher than the first, the third a value higher than the second, and so on.

Nonetheless, if you have reason to override the compiler's standard behaviour of assigning the first element an ordinal value of 0, the second a value of 1, and so on, you can. The syntax is akin to declaring a constant:

```
type
```

```
  TFunkyEnum = (OutOfSight = 64, IGotYou, GetUpIFeel = 70);
```

Any elements not explicitly assigned will be given the previous element's ordinal value plus 1. Thus, `Ord(IGotYou)` will return 65.

To get the ordinal value of an element in code, the `Ord` function is used; to go the other way, a hard cast is employed: TElemName(IntValue). The cast is 'hard' because compiler will not perform any range checking for you — `TFunkyEnum(1)` will still compile, for example, even though no valid element of `TFunkyEnum` has an ordinal value of 1.

Nonetheless, if the ordinal values of the enumeration are continuous — as they were in the `TDayOfWeek`, `TStarRating` and `TResponsibilityLevel` examples above — explicitly coding ranging checking isn't too hard:

```
function TryIntToDayOfWeek(OrdValue: Integer;
  var Day: TDayOfWeek): Boolean;
var
  Value: TDayOfWeek;
begin
  Value := TDayOfWeek(OrdValue);
  Result := (Value >= Low(TDayOfWeek)) and (Value <= High(TDayOfWeek));
  if Result then Day := Value;
end;
```

In fact, with a little help from runtime-type information (RTTI), this can be done generically:

```
uses
  System.TypInfo;

type
  TEnumConvertor<T: record> = record
    class function TryFromInteger(OrdValue: Integer;
      var EnumValue: T): Boolean; static;
  end;

class function TEnumConvertor<T>.TryFromInteger(
  OrdValue: Integer; var EnumValue: T): Boolean;
var
  Info: PTypeInfo;
  Data: PTypeData;
begin
  Info := TypeInfo(T);
  //check we have a valid type!
  if (Info = nil) or (Info.Kind <> tkEnumeration) then
    raise EArgumentException.Create(
      'TEnumConvertor not instantiated with a valid type');
  //do the actual range test
  Data := GetTypeData(Info);
  Result := (OrdValue >= Data.MinValue) and (OrdValue <= Data.MaxValue);
  if Result then Move(OrdValue, EnumValue, SizeOf(T));
end;

//...

var
  Day: TDayOfWeek;
  Rating: TStarRating;
begin
  if TEnumConvertor<TDayOfWeek>.TryFromInteger(1, Day) then
    WriteLn('1 is valid for TDayOfWeek')
  else
    WriteLn('1 in invalid for TDayOfWeek');
  //output: 1 is valid for TDayOfWeek

  if TEnumConvertor<TStarRating>.TryFromInteger(99, Rating) then
    WriteLn('99 is valid for TStarRating')
  else
    WriteLn('99 is invalid for TStarRating');
  //output: 99 is invalid for TStarRating
```

Unfortunately, such code will not work for enumerations with non-continuous ordinal values like our `TFunkyEnum`. The best you can do for that sort of case is to explicitly code something like the following on a type-by-type basis:

```
function TryIntToFunkyEnum(OrdValue: Integer;
  var Track: TFunkyEnum): Boolean;
var
  Value: TFunkyEnum;
begin
  Value := TFunkyEnum(OrdValue);
  Result := Value in [OutOfSight, IGotYou, GetUpIFeel];
  if Result then Track := Value;
end;
```

The square brackets here employ the concept of a ‘set’, which we will be looking at shortly: if `Value` is either `OutOfSight`, `IGotYou` or `GetUpIFeel`, the test succeeds.

Converting to/from string values

Aside from converting to and from an integer, another common sort of conversion is to and from a string. One way to convert *to* a string is to use a string constant. This is due to the fact static arrays support enumerated type indexers:

```
const
  DayNames: array[TDayOfWeek] = ('Sun', 'Mon', 'Tues', 'Weds',
    'Thurs', 'Fri', 'Sat');

var
  Day: TDayOfWeek;
begin
  Day := Mon;
  WriteLn(Day); //outputs Mon
end.
```

Another way is to use RTTI again, specifically the `GetEnumName` and `GetEnumValue` functions of the `System.TypInfo` unit:

```
function GetEnumName(ATypeInfo: PTypeInfo;
  AValue: Integer): string;
function GetEnumValue(ATypeInfo: PTypeInfo;
  const AName: string): Integer;
```

The value for the `ATypeInfo` argument is retrieved by calling the `TypeInfo` standard function, passing the relevant type identifier: `TypeInfo(`TEnumName`)`. `GetEnumName` then takes an element’s *ordinal* value as its second parameter; if an out-of-range value is passed in, an empty string is returned. Lastly, `GetEnumValue` returns the matching element’s ordinal value, or if no match was found, -1 (the test is case insensitive):

```
uses
  System.TypInfo;

var
  OrdValue: Integer;
  S: string;
begin
  OrdValue := GetEnumValue(TypeInfo(TDayOfWeek), 'Rubbish');
  WriteLn(OrdValue);                                   //output: -1
  OrdValue := GetEnumValue(TypeInfo(TDayOfWeek), 'MON');
  WriteLn(OrdValue);                                   //output: 1
  S := GetEnumName(TypeInfo(TDayOfWeek), OrdValue); //'Mon'
  S := GetEnumName(TypeInfo(TDayOfWeek), 999);      //''
```

This code might be wrapped into further methods on the helper type (`TEnumConvertor`) defined previously:

```
type
  TEnumConvertor<T: record> = record
    //TryFromInteger as before
    class function TryFromInteger(OrdValue: Integer;
      var EnumValue: T): Boolean; static;
    //new methods
    class function TryFromString(const Str: string;
      var EnumValue: T): Boolean; static;
    class function ToString(Value: T): string; static;
  end;

class function TEnumConvertor<T>.TryFromString(const Str: string;
  var EnumValue: T): Boolean;
var
  Info: PTypeInfo;
  IntValue: Integer;
begin
  IntValue := GetEnumValue(TypeInfo(T), Str);
  Result := (IntValue <> -1);
  if Result then Move(IntValue, EnumValue, SizeOf(T));
end;

class function TEnumConvertor<T>.ToString(Value: T): string;
var
  IntValue: Integer;
begin
```

```
  IntValue := 0;
  Move(Value, IntValue, SizeOf(T));
  Result := GetEnumName(TypeInfo(T), IntValue)
end;
```

Using the helper type, string conversion now become strongly typed, as would usually be expected in Delphi:

```
Write('Is the string ''rubbish'' a valid TDayOfWeek value?');
WriteLn(TEnumConvertor<TDayOfWeek>.TryFromString('rubbish', Day));

Write('Is the string ''MON'' a valid TDayOfWeek value? ');
WriteLn(TEnumConvertor<TDayOfWeek>.TryFromString('MON', Day));

Write('The first day of the week is ');
WriteLn(TEnumConvertor<TDayOfWeek>.ToString(Low(TDayOfWeek)));
//output: FALSE, TRUE and Mon
```

Sets

A Delphi/Pascal 'set' is an unordered group of values stemming from a single ordinal type. One popular use of set types in Delphi is as a neater alternative for a series of individual `Boolean` values. For example, the `TOpenDialog` class in the VCL uses a single `Options` set property to cover over twenty different options:

```
type
  //base enumerated type
  TOpenOption = (ofReadOnly, ofOverwritePrompt, ofHideReadOnly, ofNoChangeDir, ofShowHelp, ofNoValidate, ofAllowMultiSel

  //defines a set type based on TOpenOption
  TOpenOptions = set of TOpenOption;

  TOpenDialog = class(TCommonDialog)
  ...
  published
    property Options: TOpenOptions read ...
  end;
```

Here, the base type — i.e., what the set type is a set *of* — is an enumeration. This is the most common sort of set. You can have sets of simple integers too though, albeit within certain limitations: the base range must be continuous, and have an upper bound lower than 256:

```
type
  TValidSet = set of 1..9;          //OK
  TAnotherValidSet = set of Int8;  //OK (= 'set of 0..255')
  TInvalidSet = set of 1..256;     //causes a compiler error
```

All these examples define an explicit set type. Nonetheless, it is possible to write set expressions in an 'inline' fashion too, effectively with an inferred anonymous type:

```
procedure Foo(Value: Integer);
begin
  if Value in [1, 2, 3] then
    WriteLn('Is 1, 2 or 3')
  else
    WriteLn('Is not either 1, 2 or 3');
end;
```

Adding and removing items from a set

When a set is used as either a field of a class or a global variable, it will be initialised to an empty set. This is denoted by a pair of square brackets `[]`:

```
type
  TQuality = (qlClever, qlQuick, qlResourceful, qlThorough);
  TQualities = set of TQuality;

var
  Qualities: TQualities;
begin
  if Qualities = [] then
    WriteLn('Empty set')
  else
  begin
    Qualities := [];
    WriteLn('Wasn't an empty set, but now is');
  end;
end.
```

In the case of a non-empty set, any one value can appear once and only once. If you attempt to add the same value twice, no exception is raised, but it makes no difference to the set either:

```
var
  Qualities: TQualities;
begin
  Qualities := [qlClever];
  WriteLn(Qualities = [qlClever]);     //output: TRUE
  Qualities := Qualities + [qlClever];
  WriteLn(Qualities = [qlClever]);     //output: TRUE
```

To actually add one or more values, use either the `+` operator as shown here or the `Include` standard procedure. When using `+`, the two operands are themselves sets with the result a further set; when using `Include`, you pass a set variable as its first parameter and a single element to add as its second. The original set is modified *in situ*:

```
procedure AddQualities(var A, B: TQualities);
```

```
begin
  //add qlQuick and qlClever to A using first approach
  A := Tom + [qlQuick, qlClever];
  //add quThorough to B using second approach
  Include(B, quThorough);
end.
```

A limitation of `Include` is that it only works with set variables; attempt to use it with a set property, even a read/write one, and you will get a compiler error:

```
type
  TTest = class
  strict private
    FQualities: TQualities;
  public
    property Qualities: TQualities read FQualities write FQualities;
  end;

procedure AddQualities(Obj: TTest);
begin
  Obj.Qualities := Obj.Qualities + [qlQuick]; //OK
  Include(Obj.Qualities, qlClever);           //compiler error
end.
```

To remove a value from a set, use either the `-` operator or the `Exclude` standard procedure. These have semantics equivalent to those had by `+` and `Include` respectively.

Enumerating and testing set membership

The values currently contained in a set can be enumerated using a for/in loop:

```
procedure OutputQualities(Qualities: TQualities);
const
  SQualities: array[TQuality] of string =
    ('Clever', 'Quick', 'Resourceful', 'Thorough');
var
  Quality: TQuality;
begin
  for Quality in Qualities do
    WriteLn('- ', SQualities[Quality]);
end;
```

The `in` operator can also be used to test for a single value:

```
  if qlClever in Qualities then
    WriteLn('Watch out for the clever ones!');
```

This also works with an 'inline' set expression:

```
  if SomeIntVal in [2, 7, 10] then
    WriteLn('SomeIntVal is 2, 7 or 10');
```

To test whether two set expressions contain (or don't contain) the same values, use the equality (or inequality) operator (i.e., = or <>). The >= operator can also be used to test whether the set expression on the left contains *at least* the values of the set expression on the right, and <= can be used to test whether the set on the left is either a subset or equal to the one on the right:

```
var
  QualitiesOfSam, QualitiesOfBob: TQualities;
begin
  QualitiesOfSam := [qlClever, qlResourceful, qlThorough];
  QualitiesOfBob := [qlResourceful, qlThorough];

  Write('Does Sam have exactly the same qualities as Bob? ');
  WriteLn(QualitiesOfSam = QualitiesOfBob);              //FALSE

  if QualitiesOfSam <= QualitiesOfBob then               //no
    WriteLn('Sam has no quality beyond what Bob has.');

  if QualitiesOfSam >= QualitiesOfBob then               //yes
  begin
    WriteLn('Sam has at least the qualities Bob has.');
    if QualitiesOfSam <> QualitiesOfBob then             //yes
      WriteLn('In fact, Sam has more qualities than Bob.');
  end;
end.
```

Slightly annoyingly, the greater-than and smaller-than operators (i.e., > and <) don't work on their own. Instead, you must take two steps, first using >= or <= before testing for inequality separately, as shown here.

The final operator that works with set types is `*`. This returns the intersection of its two operands, i.e., a new set that contains the values had by both the two source sets:

```
WriteLn('Sam and Bob''s shared qualities are the following:');
OutputQualities(QualitiesOfSam * QualitiesOfBob);

WriteLn('Out of being clever and/or resourceful, Bob is:');
OutputQualities(QualitiesOfBob * [qlClever, qlResourceful]);
```

Manual 'sets': bitwise operators

There are a few places in the Delphi RTL — and many, many places in the Windows API — where 'bitmasks' on an integer of some sort are used instead of proper set types. For example, the `System.SysUtils` unit declares a function, `FileGetAttr`, that returns the attributes of a file (read only, hidden, system, etc.) in the form of a single integer. To test for individual attributes, you must use so-called 'bitwise' operators, in particular `and`:

```
uses
  System.SysUtils;

var
  Attr: Integer;
begin
  Attr := FileGetAttr('C:\Stuff\Some file.txt');
  if Attr and faReadOnly <> 0 then
    WriteLn('Read only');
  if Attr and faHidden <> 0 then
    WriteLn('Hidden');
  if Attr and faSysFile <> 0 then
    WriteLn('System');
end.
```

For this to work, the `faxxx` constants are defined so that the first has a value of 1, the second a value of 2, the third a value of 4, the fourth of 8 and so on. To add a value to an existing manual 'set', use `or`, and to remove a value, use `and not`:

```
procedure AddHiddenAttr(const AFileName: string);
begin
  FileSetAttr(AFileName, FileGetAttr(AFileName) or faHidden);
end;

procedure RemoveHiddenAttr(const AFileName: string);
begin
  FileSetAttr(AFileName, FileGetAttr(AFileName) and not faHidden);
end;
```

In general, you should use proper set types where you can, since they provide strong typing and greater readability. Nonetheless, the following code demonstrates the fact that under the bonnet, 'real' sets and their manual, C-style equivalents boil down to the same thing:

```
const
  mcFirst  = 1;
  mcSecond = 2;
  mcThird  = 4;
  mcFourth = 8;

type
  TMyEnum = (meFirst, meSecond, meThird, meFourth);
  TMySet = set of TMyEnum;

var
  UsingSet: TMySet;
  UsingConsts: LongWord;
begin
  //direct assignment
  UsingSet := [meSecond, meThird];
  UsingConsts := mcSecond or mcThird;
  WriteLn('Equal? ', Byte(UsingSet) = UsingConsts);
  //subtraction
  Exclude(UsingSet, meSecond);
  UsingConsts := UsingConsts and not mcSecond;
  WriteLn('Equal? ', Byte(UsingSet) = UsingConsts);
  //addition
  Include(UsingSet, meFourth);
  UsingConsts := UsingConsts or mcFourth;
  WriteLn('Equal? ', Byte(UsingSet) = UsingConsts);
  //membership test
  if meThird in UsingSet then WriteLn('meThird is in');
  if UsingConsts and mcThird <> 0 then WriteLn('mcThird is in');
end.
```

Run the program, and you’ll find TRUE outputted in each case.

Working with numbers: integers

In most circumstances, if you need to use *an* integer type, you just should use *the* `Integer` type. However, when you need a guaranteed size for (say) interoperability reasons, Delphi directly supports integers that are 8, 16, 32 and 64 bits wide, in both signed and unsigned forms:

Type name	Size in bytes	Range	Standard aliases
Byte	1	0 to 255	UInt8
Word	2	0 to 65,535	UInt16
LongWord	4	0 to 4,294,967,295	UInt32, DWORD
UInt64	8	0 to (2^{64})-1	None
ShortInt	1	-128 to 127	Int8
SmallInt	2	-32,768 to 32,767	Int16
LongInt	4	-2,147,483,648 to 2,147,483,647	UInt32
Int64	8	-2^{63} to (2^{63})-1	None

Since the base names can be confusing — a `SmallInt` is different from a `ShortInt`, and `LongInt` is not in fact the 'longest' type — it is generally preferable to use the `IntX`/`UIntX` aliases instead.

Aside from `Integer`, which is the main generic signed type, `Cardinal` is defined as the generic *un*signed type, i.e. starting at zero rather than a negative figure. Both `Integer` and `Cardinal` map to 32 bit types at present. This means that while you can cast to either from a pointer when using the 32 bit compiler, you cannot when using the 64 bit compiler. (`Integer` and `Cardinal` stay 32 bit since that is the model Microsoft chose for Win64.) Because of that, further generic types are defined, `NativeInt` and `NativeUInt`, which are guaranteed to be typecastable with pointers whatever the target platform. In practical terms, this means you can use the `Tag` property of a VCL control to store a reference to another object, despite `Tag` itself being an integer not an object property:

```
LeftHandEdit.Tag := NativeInt(RightHandEdit);
//...
if TEdit(LeftHandEdit.Tag).Text <> '' then //...
```

It is usually a better idea to avoid typecasting hacks like this though — in the current case, using a strongly typed `TDictionary` (a class we will be looking at in chapter 6) would be a much better way of mapping one control to another.

Integer literals

Integer literals in Delphi are as you would expect — just type the number, though don't include any commas or spaces to separate out thousands or whatever. Hexadecimal (base 16) notation can also be used by prefixing the number with a dollar sign:

```
const
  DaysPerLeapYear = 366;   //decimal (base 10)
  BytesPerKilobyte = $400; //hexadecimal (base 16)
  BakersDozen = $D;        //also hexadecimal
```

Integer operators and intrinsic routines

The core arithmetic operators in Delphi are `+` for addition, `-` for subtraction, `*` for multiplication, `mod` for the remainder after a division has been performed, `/` for division that returns a *real* (non-whole) number, and `div` for division that returns another integer, rounding down if necessary. If you use `/` rather than `div`, then even if both input values are integers and the result will be another number without remainder, the result will not be directly assignable to another integer:

```
var
  I1, I2, I3: Integer;
begin
  I1 := 20;
  I2 := 2;
  I3 := I1 div I2; //OK
  I3 := I1 / I2;   //compiler error
```

The group of intrinsic routines we met when looking at enumerated types work with integers too. Thus. Thus, `Inc(X)` is an alternative way of writing `X := X + 1`, `Dec(X, 3)` is an alternative way of writing `X := X - 3`, `Succ(X)` returns `X + 1`, and `Pred(X)` returns `X - 1`.

For comparison, `=` tests for equality, `<>` for inequality, `<` for a first value being less than a second, `>` for it being greater, `<=` for it being smaller or equal, and `>=` for it being bigger or equal. In all cases, a `Boolean` results:

```
var
  I1, I2, Counter: Integer;
begin
  Counter := 0;
  I1 := 23;
  I2 := 12;
  if I1 > I2 then Inc(Counter);
```

Commonly-used integer functions

The `System` unit declares a small number of routines that work with integers: `Abs`, `Random`, `Trunc` and `Round`. `Abs` strips off any sign:

```
Num := Abs(-4); //assigns 4
Num := Abs(4);  //also assigns 4
```

`Random` returns a pseudo-random number. If no argument is passed, a real (i.e., non-whole) number between 0 and 1 is returned, otherwise an integer between 0 and the number you pass it, minus one, is:

```
RealNum := Random; //return a no. 0 <= x < 1
Num := Random(11); //return a random no. between 0 and 10
```

Call `Randomize` (preferably only once, at program start up) to ‘seed’ the random number generator using the system time; you can also provide a specific seed by assigning the `RandSeed` global variable. This allows returning the same sequence of ‘random’ numbers each time.

Since integers and ‘real’ numbers are different types, you cannot simply assign a number that contains a decimal point to an integer. Instead, call either `Trunc` to simply truncate the number, or `Round` to round it. Since `Round` uses ‘banker’s rounding’, if the number passed to it is exactly half way between two whole numbers, the nearest even number is returned:

```
const
  RealNo = 4.5;
var
  Int: Integer;
begin
  Int := Trunc(RealNo);            //4
  Int := Round(RealNo);            //also 4 (nearest whole no.)
  Int := Round(RealNo + 0.000001); //5
end;
```

Add `System.Math` to your uses clause, and a number of other useful functions will be available:

- `Ceil` rounds a real number up, `Floor` rounds down (`Trunc` will only do likewise when the input is not negative).
- `Min` returns the smallest value amongst two numbers, `Max` returns the biggest value amongst two numbers, `MinIntValue` the smallest number from an array of integers, and `MaxIntValue` the biggest number from an array of integers:

```
  I := Min(3, 9);               //assigns 3
  I := MaxIntValue([9, 87, 2]); //assigns 87
```

- `InRange` returns a `Boolean` indicating whether a given number is within a specified range, inclusive:

```
  OK := InRange(4, 2, 5); //TRUE
  OK := InRange(4, 4, 5); //TRUE
  OK := InRange(4, 2, 4); //TRUE
  OK := InRange(4, 5, 9); //FALSE
```

- `EnsureRange` takes three numbers, returning the first if it is within the bounds specified by the second and third, the second if it is lower and the third if it is higher:

```
  I := EnsureRange(8, 2, 11); //8
  I := EnsureRange(8, 9, 11); //9
  I := EnsureRange(8, 2, 5);  //5
```

- Building on `Random`, `RandomRange` returns a 'random' number in between two figures inclusive; similarly, `RandomFrom` returns a 'random' item from an array of numbers:

```
const
  ArrayConst: array[1..4] = (8, 6, 2, 9);
var
  NumDynArray: TArray<Integer>;
  Picked: Integer;
begin
  Randomize;                          //Call this once only
  Picked := Random(45);               //No. less than 45
  Picked := RandomRange(50, 90);      //No. >=50 and <=90
  Picked := RandomFrom([2, 6, 4, 5]); //No. from 2, 6, 4 and 5
  Picked := RandomFrom(ArrayConst);   //No. from 8, 6, 2 and 9
  NumDynArray := TArray<Integer>.Create(63, 5, 7);
  Picked := RandomFrom(NumDynArray);  //No. from 63, 5 and 7
```

Converting from an integer to a string

To convert an integer to a string, use `IntToStr` for signed integer types and `UIntToStr` for unsigned ones:

```
var
  SignedInt: Integer;
  BigUnsignedInt: UInt64;
  S: string;
begin
  SignedInt := -10;               //assigns '-10'
  S := IntToStr(SignedInt);
  BigUnsignedInt := SignedInt + 100000000000;
  S := UIntToStr(BigUnsignedInt); //assigns '99999999990'
```

These all convert to decimal notation; to convert to hexadecimal instead, use `IntToHex`. This takes two arguments, the integer to convert and the minimum number of digits to return. The string returned will be without any prefix indicating it is a hexadecimal number:

```
var
  SignedInt: Integer;
  BigUnsignedInt: UInt64;
  S: string;
begin
  SignedInt := 20;
  S := IntToHex(SignedInt, 4);
  WriteLn(S);                       //output: 0014
  BigUnsignedInt := SignedInt + 1234567890;
  S := IntToHex(BigUnsignedInt, 4);
  WriteLn(S);                       //output: 499602E6
```

Also useful is the `Format` function, as declared in the `System.SysUtils` unit. Converting more than just integers, this takes a 'format string' and an array of values to slot into it. The places they slot into are determined by codes embedded in the format string:

```
S := Format('Slot in an integer: %d', [SomeInt]);
```

Each code begins with a percent sign; in the simplest cases, this is then immediately followed by a single letter that denotes the type. In the case of integers, the relevant letters are **d**, **u** and **x**:

- **d** (for 'decimal') behaves like `IntToStr`:

  ```
  S := Format('It is %d', [65]); //assigns 'It is 64'
  ```

  Using a 'precision specifier', you can require a minimum number of digits. This is done by placing a full stop or period and the number in between the percent sign and `d`:

  ```
  S := Format('%.4d', [1]); //assigns '0001'
  ```

- **u** ('unsigned') is like **d**, but works with unsigned input. If a signed integer is passed, it is simply cast to an unsigned integer. Given the way a signed integer is represented internally, this will *not* just strip off the sign from a lay person's point of view. E.g., `Format('%u', [-1])` returns `'4294967295'`, not `'1'`. If just stripping off the sign is what you want, then pass the value to the `Abs` standard function first: `Format('%u', [Abs(-1)])`.

- **x** behaves as `IntToHex`, only where setting the number of digits (now done via a precision specifier) is optional:

  ```
  S := Format('%x', [10]);   //assigns 'A'
  S := Format('%.4x', [10]); //assigns '000A'
  ```

  To include a dollar sign or any other prefix, just embed it in the format string:

```
  S := Format('$%.4x', [10]); //assigns '$000A'
```

Also useful for integers are **n** and **s**:

- **n** (for 'number') takes a real number and converts it to a string using thousand separators, which are things **d** (like `IntToStr`) never includes. Since **n** expects a real number, you must call the `Int` intrinsic function to avoid an exception. Furthermore, since **n** defaults to outputting to 2 decimal places, you must pass 0 as a precision specifier to force none:

```
  S := Format('%n', [Int(5999000)]);   //assigns '5,999,000.00'
  S := Format('%.0n', [Int(5999000)]); //assigns '5,999,000'
  S := Format('%n', [5999000]);        //runtime exception!
```

- **s** ('string') takes a string input:

```
  function GetDesc(const FileName: string; KB: Int64): string;
  begin
    Result := Format('The file "%s" is %.0nKB', [FileName, KB)]);
  end;
```

Converting from a string to an integer

For converting from a string to an integer, `System.SysUtils` provides a choice of `StrToInt`, `StrToIntDef` and `TryStrToInt`, which all return 32 bit integers, and `StrToInt64`, `StrToInt64Def` and `TryStrToInt64`, which all return 64 bit ones. The difference between the three in each set is what happens when the string cannot be converted to an integer: `StrToInt` and `StrToInt64` will raise an `EConvertError` exception, `StrToIntDef` and `StrToInt64Def` will return a specified default value, and `TryStrToInt` and `TryStrToInt64` will return `False`:

```
var
  S: string;
  Value: Integer;
begin
  Value := StrToInt('23');
  Value := StrToIntDef('rubbish', 99); //returns 99
  if TryStrToInt('$AC', Value) then    //succeeds
    WriteLn('OK - ', Value)
  else
    WriteLn('Not OK');
```

The string passed can contain leading spaces (e.g. ' 23'), and can be a hexadecimal number if it is prefixed with either a dollar sign (Delphi/Pascal-style) or a zero followed by an X (C-style), or just an X (the X can be in either lower or upper case). However, attempting to convert a string with trailing spaces and/or thousand separators will fail:

```
Value := StrToIntDef('4,000', 99); //returns 99
Value := StrToIntDef('4000 ', 99); //also returns 99
```

If this is an issue, strip the spaces and separators out first:

```
function StripThousandSepsAndTrim(const S: string): string;
begin
  Result := StringReplace(Trim(S),
    FormatSettings.ThousandSeparator, '', [rfReplaceAll]);
end;

begin
  Value := StrToIntDef(StripThousandSepsAndTrim('4,000 '), 99);
```

Like `StrToIntDef` itself, `Trim` and `StringReplace` are functions provided by `System.SysUtils`; declared in the same unit, `FormatSettings` is a global record variable which contains the localised format settings for the computer the program is being run on. If you want to always strip out commas regardless of whether it is the active separator, just call `StringReplace` twice:

```
function StripThousandSepsAndTrim(const S: string): string;
begin
  Result := StringReplace(Trim(S),
    FormatSettings.ThousandSeparator, '', [rfReplaceAll]);
  if FormatSettings.ThousandSeparator <> ',' then
    Result := StringReplace(Result, ',', '', [rfReplaceAll]);
end;
```

Sub-range types

All the built in integer types accept the widest range of values logically possible for their size and signage. You can however define 'sub-range' types if you want a more restricted range. Their general form is

```
type
  Ident = MinValue..MaxValue;
```

In fact, sub-range types can be based on *any* ordinal type, though basing on an integer is commonest:

```
type
  TCapitalAsciiLetter = 'A'..'Z';
  TStarRating = 0..5;
  TCarDoors = 2..5;

var
  Rating: TStarRating;
  NumOfDoors: TCarDoors;
  Letter: TCapitalAsciiLetter;
begin
  Rating := 1;     //OK
  NumOfDoors := 3; //OK
  Rating := 6;     //won't compile
  NumOfDoors := 7; //won't compile
  Letter := 'G';   //OK
  Letter := 'a';   //won't compile
end.
```

Sub-range types can also be declared inline if you wish:

```
var
  InlineExample: 100..199;
```

In either case, the underlying type used will be smallest standard type that can accept the stated bounds; in all but the `TCapitalAsciiLetter` example above, that will be `Byte`.

Unfortunately, the default compiler settings make it easy to assign an out-of-range value to a sub-range type. All you need to do is to assign the value to a variable of the base type first:

```
var
  NumOfDoors: TCarDoors;
begin
  IntValue := 99;
  NumOfDoors := IntValue; //assigns without complaint
  WriteLn(NumOfDoors);    //outputs 99
end.
```

In practice, the utility of sub-range types is therefore greatest when ‘range-checking’ is enabled.

Range and overflow checking

In Delphi, forcing integer values within bounds involves two different compiler directives, `$OVERFLOWCHECKS` (`$Q` for short) and `$RANGECHECKS` (`$R`). Both can be set globally (e.g. as part of a particular build configuration in the IDE), and both can be set locally too, integrated into the source code:

```
{$OVERFLOWCHECKS ON}  //enable overflow checking from here
{$RANGECHECKS ON}     //enable range checking from here

{$OVERFLOWCHECKS OFF} //disable overflow checking from here
{$RANGECHECKS OFF}    //disable range checking from here
```

When range checking is enabled, assignments from wider to narrower types are automatically checked; out of range values for the destination type when they happen cause an `ERangeError` to be raised. This makes the `TCarDoors` code snippet just presented now work as you would expect.

Range checking also works with `Inc` and `Dec` calls:

```
{$RANGECHECKS ON}
var
  Value: 1..10;
begin
  Value := 10; //assign the maximum value
  Inc(Value);  //causes an ERangeCheck error
```

However, if overflow checking isn’t enabled too, things can still seem strange to the uninitiated:

```
{$OVERFLOWCHECKS OFF}
{$RANGECHECKS ON}
type
  TMySubRangeType = 1..High(Byte);
var
  SubRangeValue: TMySubRangeType;
begin
  try
    SubRangeValue := High(Byte); //assign the maximum value
    Inc(SubRangeValue, 2);       //add 2
    WriteLn(SubRangeValue);
  except
    on E: Exception do WriteLn(E.ClassName, ': ' + E.Message);
  end;
end.
```

Here, no exception is raised and 1 is outputted. The lack of an exception is because when overflow checking is not enabled, incrementing beyond the highest possible value causes the value to 'wrap'. Change `{$OVERFLOWCHECKS OFF}` to `{$OVERFLOWCHECKS ON}`, and an `EIntOverflow` exception would be raised.

Even taking account of the wrapping however, it may seem odd that 1 is returned given the second value of `TMySubRangeType` is 2. The reason is that the wrapping is performed on the base intrinsic type (here, `Byte`) *before* the range check is performed. One consequence of this is that if the code were changed so that `SubRangeValue` is incremented by 1 rather than 2, an exception would be raised even without overflow checking being enabled!

In general, it is a good idea to enable range checking, even for release builds. Matters are a bit more ambiguous in the case of overflow checking though, since an assignment that would pass a range check yet fail an overflow one would result in a value still formally valid for the stated type. Consequently, it is perfectly possible to write code that *expects* overflow checking disabled in order to work properly.

When such a situation arises however, a common idiom is to explicitly disable overflow checking before re-enabling it as appropriate. This is made possible by the `$IFOPT` compiler directive:

```
implementation

{$IFOPT Q+}                     //Is overflow checking enabled?
  {$DEFINE OverflowChecksOn} //Set a 'conditional define' if it
{$ENDIF}                        //is (name used is arbitrary).

{ Other things can be here... }

{$Q-}                           //Disable overflow checking.
procedure KnowsWhatItsDoing;
begin
  { Do stuff assuming overflow checking is disabled... }
end;
{$IFDEF OverflowChecksOn}{$Q+}{$ENDIF} //Renable if applicable
```

Once set, a conditional define will be valid for the rest of the unit. The point of setting it in the first place is to respect whatever setting is made globally — if you always compile with overflow checks on, you can just do this instead:

```
implementation

{$OVERFLOWCHECKS OFF}
procedure KnowsWhatItsDoing;
begin
  (* do stuff assuming overflow checking is disabled... *)
end;
{$OVERFLOWCHECKS ON}
```

Real numbers: floating point types

A 'real' number is one that may involve something after the decimal point; a 'floating point type' ('float' for short) is the way real numbers are usually represented by a computer. Following the original Pascal language, Delphi defines a generic floating point type called `Real` whose precise implementation may vary across compiler versions. Similar to the situation with integers, a small set of 'fundamental' types stands next to the generic one:

Type name	Size in bytes	Significant digits	Notes
Single	4	7-8	IEEE 754 standard
Real48	6	11-12	Legacy type
Double	8	15-16	IEEE 754 standard
Extended	8/10	10-20/15-16	See below
Extended80	10	10-20	Interoperability type
Currency	8	10-20	Fixed to 4 decimal places

The greater the size of a float, the more numbers it can accurately and not just approximately represent.

Legacy vs. non-legacy types

Of the floating point types Delphi supports, `Real48` is a compatibility type for very old code and shouldn't be used unless parsing files written using either very old versions of Delphi or its predecessor, Turbo Pascal. `Extended` is also a legacy type to an extent: whilst a proper 10 byte `Extended` type is native to the 32 bit platforms Delphi supports, `Extended` is simply mapped to the 8 byte `Double` when compiling for 64 bit. This may sound a bit weird — a type with greater range is lost when the bitness of the target platform increases — but that's just how it is.

Nonetheless, since this loss can cause issues for old code that wrote `Extended` values out to files, `Extended80` type is defined as a compatibility type. When compiling for 32 bit, it maps to `Extended`. However, when compiling for 64 bit, it acts as a special container type, holding 10 byte values without implementing 10 byte-standard arithmetic.

In contrast to `Real48` and `Extended`, `Single` and `Double` are native types on all platforms Delphi supports. Of the two, `Double` is best for general use given the `Single` type's relative lack of precision, however complex graphics programming (where precision is less of an issue than memory) can be an exception.

Floating point literals

Floating point literals can be entered using either normal decimals or scientific 'e' notation:

```
const
  WeeksPerYear = 52.143;
  EarthMass = 5.9736e24; //i.e., 5.9736 x 10 to the power of 24
```

In the case of an integral literal, you can force a specific type with a cast. This is not possible with floats — literals are fixed to the `Extended` type with the 32 bit compiler and `Double` when targeting 64 bit Windows. If the type is important, you must therefore use a typed constant instead, though this does come at the cost of not allowing the 'constant' to enter into a constant expression:

```
const
  TypedConst: Single = 123.456; //OK, as really a read-only var
  TrueConst = Single(123.456);  //doesn't compile
```

The Currency type

By their very nature, floating point types involve rounding errors — very small rounding errors, but errors that can become untenable in complex financial calculations as they progressively accumulate. The `Currency` type exists as one way to work around the issue. It works on the principle that if rounding is needed, it should be done as you go along; consequently, a `Currency` value is implemented as a scaled 64 bit integer, a value of 1.1 being stored as 11,000, a value of 123.4567 stored as 1,234,567 and so on.

The following program demonstrates the good and the bad implications of this:

```
uses
```

```
  System.SysUtils;

var
  I: Integer;
  CurrValue: Currency;
  DblValue: Double;
begin
  CurrValue := 0;
  DblValue := 0;
  for I := 1 to 90 do
  begin
    CurrValue := CurrValue + 0.1;
    DblValue := DblValue + 0.1;
  end;
  WriteLn(CurrToStr(CurrValue));
  WriteLn(FloatToStr(DblValue));
  WriteLn;
  CurrValue := 1;
  DblValue := 1;
  for I := 1 to 5 do
  begin
    CurrValue := CurrValue * 0.1;
    DblValue := DblValue * 0.1;
  end;
  WriteLn(CurrToStr(CurrValue));
  WriteLn(FloatToStr(DblValue));
end.
```

After adding up ninety 0.1's, the `Currency` output is 9 where the `Double` output is 8.99999999999998. So, score one for `Currency`. However, after multiplying 1 by 0.1 five times, the `Currency` output is 0! This is due to the rounding to four decimal places. In contrast, the `Double` output is 1E-5 (i.e., 0.0000000001).

This goes to show `Currency` isn't a general solution for floating point woes. If you have a situation where only four decimal places is fine, then it is ideal, otherwise it will be a non-starter.

Special floating point values

With the exception of the `Currency` type, floats can have the special values 'not a number' (NaN), infinity and negative infinity. To assign them, use the `NaN`, `Infinity` and `NegInfinity` constants, as defined by the `System.Math` unit. To test for them, use the `IsNaN` and `IsInfinite` functions of the same unit. If you try to test for them directly, you will fail:

```
Val := NaN;
WriteLn(IsNan(Val)); //outputs FALSE
WriteLn(Val = NaN);  //EInvalidOp - Invalid floating pt operation
```

If a calculation is performed in which one of the values is a NaN, then NaN will be returned — in a word, NaNs 'propagate'. Similarly, when an infinite value is part of an expression, then infinity is returned:

```
Value := NaN;
Value := Value * 2 + 23.9;
WriteLn(FloatToStr(Value)); //outputes NAN
Value := NegInfinity;
Value := Value + 59.27;
WriteLn(FloatToStr(Value)); //outputes -INF
```

If both infinity *and* a NaN is part of an expression, then NaN takes priority.

Dealing with imprecision

If you try to compare two floating point values naïvely, you are likely to get surprising results. For example, the following outputs FALSE:

```
var
  Value: Double;
begin
  Value := 0.01;
  WriteLn('Does Value = 0.01? ', Value = 0.01);
```

The solution is to allow a small amount of leeway whenever you compare two floats. While you can do it yourself, the `SameValue` function from `System.Math` calculates an appropriate amount of leeway for you:

```
var
  Value: Double;
begin
  Value := 0.01;
  WriteLn('Does Value = 0.01? ', SameValue(Value, 0.01)); //TRUE
```

`IsZero` is also available to test for being either zero or very, very close to it following a series of calculations.

The reason the problem arises in the first place is twofold. Most fundamentally, a float cannot hope to represent *every* possible real number, since the range of possible reals forms a continuum from negative to positive infinity (as 2.45 lies between 2.4 and 2.5, so 2.455 lies between 2.45 and 2.5, 2.4555 lies between 2.455 and 2.5, and so on). Secondly, the fact computers think in binary (base 2) rather than decimal (base 10) leads to finite numbers on paper not necessarily being finite numbers in base 2, and therefore, not perfectly representable by a floating point type. An example of this is 0.1, and in general, only numbers that can be calculated by summing powers of 2 can be stored with complete accuracy in a float. This isn't a Delphi-specific problem — you will find it arising with whatever computer language you may come to use.

Performance vs. precision: 32 bit vs. 64 bit compilation

If you are performing a lot of floating point calculations and performance is important, using the 64 bit compiler should give your code a big boost, due to the fact it uses a more modern CPU instruction set for floating point compared to the 32 bit compiler. However, this comes at the cost of less precision; aside from the fact the 32 bit compiler supports a proper `Extended` type where the 64 bit one does not, calculations under 32 bit are performed using `Extended` precision regardless of the input or output types.

If the 64 bit compiler didn't do anything special, it would instead have `Single` precision used when performing calculations on `Single` values, and `Double` precision used when performing calculations on `Double` values, and indeed, this is exactly what happens in the `Double` case. In the `Single` one though, the compiler's default behaviour is to make up for the loss of precision by using hidden `Double` intermediaries whenever two `Single` values enter into an expression to calculate a third. While this is a reasonable default from a backwards compatibly standpoint (Microsoft's 64 bit C++ compiler does the same thing, and for the same reason), it has the effect of making `Single` arithmetic even slower with the 64 bit compiler than it is with the 32 bit one!

Happily, this counter-intuitive result can be corrected easily though — just set the `$EXCESSPRECISION` compiler option to `OFF` in code:

```
{$EXCESSPRECISION OFF}
```

Add this line either to the top of any unit where lots of complex `Single` calculations are performed, or alternatively, in the DPR underneath any existing directives, and `Single` arithmetic will now be performed at full pelt the next time you build your application.

Cracking open a floating point value

Floating point numbers have a complex internal representation, being made up of specific bits for the sign, 'mantissa' and 'exponent', together with and a few other things to allow for infinity and NaN states. In order to make it easier to inspect the various components of a floating point value, the `System` unit provides a set of record types, one each for `Single`, `Double` and `Extended80`. The one for `Double` looks this (private members have been removed for clarity):

```
type
  TFloatSpecial = (fsZero, fsNZero, fsDenormal, fsNDenormal,
    fsPositive, fsNegative, fsInf, fsNInf, fsNaN );

  PDoubleRec = ^TDoubleRec;
  TDoubleRec = packed record
    function Exponent: Integer;
    function Fraction: Extended;
    function Mantissa: UInt64;

    property Sign: Boolean read GetSign write SetSign;
    property Exp: UInt64 read GetExp write SetExp;
    property Frac: UInt64 read GetFrac write SetFrac;

    function SpecialType: TFloatSpecial;
    procedure BuildUp(const SignFlag: Boolean;
      const Mantissa: UInt64; const Exponent: Integer);

    case Integer of
      0: (Words: array [0..3] of UInt16);
      1: (Bytes: array[0..7] of UInt8);
  end;
```

It can be used by casting a `Double` value to a `TDoubleRec`:

```
const
```

```
  Types: array[TFloatSpecial] of string = ('zero',
    'negative zero', 'denormal', 'negative denormal', 'positive',
    'negative', 'infinity', 'negative infinity', 'NaN');
  Signs: array[Boolean] of string = ('+', '-');
var
  Rec: TDoubleRec;
  Value: Double;
begin
  Value := 5.5;
  WriteLn(Value);
  Rec := TDoubleRec(Value);                  //make the cast
  WriteLn('Type: ', Types[Rec.SpecialType]); //positive
  WriteLn('Exponent: ', Rec.Exponent);       //2
  WriteLn('Exponent (raw): ', Rec.Exp);      //1025
  WriteLn('Fraction: ', Rec.Fraction);       //1.375
  WriteLn('Fraction (raw): ', Rec.Frac);     //1688849860263936
  WriteLn('Sign: ', Signs[Rec.Sign]);        //+
end.
```

Note that the `Sign` property reports whether the sign bit is turned on. Since positive numbers are the default, when `Sign` is `True` the value will be negative!

Details of use aside, while cracking open a floating point value can be interesting, it is rarely useful in practice.

Mathematical routines

The `System` unit provides a small set of standard routines that work on real numbers:

```
{ the three trigonometric functions all work in radians }
function ArcTan(X: Extended): Extended; //arctangent of X
function Cos(X: Extended): Extended;    //cosine of angle X
function Sin(X: Extended): Extended;    //sine of angle X

function Exp(X: Real): Real; //raises e to the power of X
function Ln(X: Real): Real;  //natural logarithm of X - Ln(e) = 1
function Pi: Extended;                //3.1415926535897932385
function Sqr(X: Extended): Extended;  //square of X
function Sqrt(X: Extended): Extended; //square root of X

function Frac(X: Extended): Extended; //Frac(1.1) = 0.1
function Int(X: Extended): Extended;  //Int(1.1) = 1.0
function Trunc(X: Extended): Int64;   //Trunc(1.1) = Int64(1)

function Random: Extended; //get a 'random' positive number < 1
```

`System.Math` adds quite a few more, with almost all overloaded with `Single`, `Double` and `Extended` versions. First up are float variants of integer functions we've met previously:

```
function Min(const A, B: Extended): Extended;
function MinValue(const Data: array of Extended): Extended;
function Max(const A, B: Extended): Extended; overload;
function MaxValue(const Data: array of Extended): Extended;
function InRange(const AValue, AMin, AMax: Extended): Boolean;
function EnsureRange(const AValue, AMin, AMax: Extended): Extended;
function RandomFrom(const AValues: array of Extended): Extended;
```

Three exponential functions follow:

```
function IntPower(const Base: Extended;
  const Exponent: Integer): Extended;
function Power(const Base, Exponent: Extended): Extended;
function Ldexp(const X: Extended; const P: Integer): Extended;
```

`IntPower` and `Power` both raise `Base` to the power of `Exponent`; `Ldexp` returns `X` multiplied by `2` to the power of `P`. For example, both `IntPower(4, 6)` and `Power(4, 6)` calculate 46, and `Ldexp(5, 9)` calculates 5 x (29).

Statistical functions

`RandG` provides a way to get 'random' numbers with a Gaussian (i.e., 'normal' or bell-shaped) distribution about a given mean value:

```
function RandG(Mean, StdDev: Extended): Extended;
```

Other statistical functions include the following:

```
function Mean(const Data: array of Extended): Extended;
function Sum(const Data: array of Extended): Extended;
function SumOfSquares(const Data: array of Extended): Extended;
```

```
procedure SumsAndSquares(const Data: array of Extended;
  var Sum, SumOfSquares: Extended);
function StdDev(const Data: array of Extended): Extended;
procedure MeanAndStdDev(const Data: array of Extended;
  var Mean, StdDev: Extended);
function PopnStdDev(const Data: array of Extended): Extended;
function Variance(const Data: array of Extended): Extended;
function PopnVariance(const Data: array of Extended): Extended;
function TotalVariance(const Data: array of Extended): Extended;
function Norm(const Data: array of Extended): Extended;
procedure MomentSkewKurtosis(const Data: array of Double;
  var M1, M2, M3, M4, Skew, Kurtosis: Extended); //no overloads
```

`Mean` returns the arithmetic average, `PopnVariance` the total variance divided by the number of items, `Variance` the total variance divided by the number of items minus one, `PopnStdDev` the square root of the population variance, and `Norm` returns the Euclidean L2-norm, i.e., the square root of the sum of squares.

As in all cases where `array of SomeTime` is part of the declaration, a static, dynamic or 'inline' array can be passed for the input data:

```
var
  DynArray: array of Double;
  StaticArray: array[1..100] of Single;
  M1, M2, M3, M4, Skew, Kurtosis: Extended;
  I: Integer;
begin
  Randomize;
  //use 'inline' array
  WriteLn('The mean of 36, 29 and 42 is ', Mean([36, 29, 42]));
  //use static array
  for I := Low(StaticArray) to High(StaticArray) do
    StaticArray[I] := Random;
  WriteLn('The sum of squares for my series of numbers is ',
    SumOfSquares(StaticArray));
  //use dynamic array
  SetLength(DynArray, 200);
  for I := Low(DynArray) to High(DynArray) do
    DynArray[I] := Random * MaxDouble;
  MomentSkewKurtosis(DynArray, M1, M2, M3, M4, Skew, Kurtosis);
  WriteLn('Using another random series of numbers...');
  WriteLn('The first moment is ', M1);
  WriteLn('The second moment is ', M2);
  WriteLn('The third moment is ', M3);
  WriteLn('The fourth moment is ', M4);
  WriteLn('The skew is ', Skew);
  WriteLn('The kurtosis is ', Kurtosis);
```

Trigonometric functions

Supplementing the trigonometric functions from `System` are many additional ones in `System.Math`, too numerous to list here (e.g. `ArcCos`, `ArcSin`, `Tan`, `Cotan`, `Secant`, etc., along with hyperbolic versions — `ArcCosH` etc.). All angles are specified in terms of radians. Conversion functions are therefore provided to convert between radians, degrees, cycles and gradients, where radians = degrees * pi / 180, one cycle is 360°, and gradients = radians * 200 / pi:

```
WriteLn('180° is the same as...');
WriteLn(FloatToStr(DegToRad(180)) + ' radians');      //3.14159
WriteLn(FloatToStr(DegToGrad(180)) + ' gradients');   //200
WriteLn(FloatToStr(DegToCycle(180)) + ' cycles');     //0.5

WriteLn('0.4 cycles is the same as...');
WriteLn(FloatToStr(CycleToRad(0.4)) + ' radians');    //2.51327
WriteLn(FloatToStr(CycleToGrad(0.4)) + ' gradients'); //160
WriteLn(FloatToStr(CycleToDeg(0.4)) + ' degrees');    //144

WriteLn('250 gradients is the same as...');
WriteLn(FloatToStr(GradToRad(250)) + ' radians');     //3.92699
WriteLn(FloatToStr(GradToCycle(250)) + ' cycles');    //0.625
WriteLn(FloatToStr(GradToDeg(250)) + ' degrees');     //255

WriteLn('1 radian is the same as...');
WriteLn(FloatToStr(RadToDeg(1)) + ' degrees');        //57.2957
WriteLn(FloatToStr(RadToGrad(1)) + ' gradients');     //63.6619
WriteLn(FloatToStr(RadToCycle(1)) + ' cycles');       //0.15915
```

Logarithmic and financial functions

Building on the `System` unit's `Ln` function, which returns the natural logarithm of a number, `System.Math` adds the following:

```
function LnXP1(const X: Extended): Extended; //natural log of X+1
function Log10(const X: Extended): Extended; //log base 10 of X
function Log2(const X: Extended): Extended;  //log base 2 of X
function LogN(const Base, X: Extended): Extended; //log base N of X
```

Lastly, a series of financial functions are declared:

```
type
  TPaymentTime = (ptEndOfPeriod, ptStartOfPeriod);

function DoubleDecliningBalance(const Cost, Salvage: Extended;
  Life, Period: Integer): Extended;
function FutureValue(const Rate: Extended; NPeriods: Integer;
  const Payment, PresentValue: Extended;
  PaymentTime: TPaymentTime): Extended;
function InterestPayment(const Rate: Extended; Period,
  NPeriods: Integer; const PresentValue, FutureValue: Extended;
  PaymentTime: TPaymentTime): Extended;
function InterestRate(NPeriods: Integer; const Payment,
  PresentValue, FutureValue: Extended;
  PaymentTime: TPaymentTime): Extended;
function InternalRateOfReturn(const Guess: Extended;
  const CashFlows: array of Double): Extended;
function NumberOfPeriods(const Rate: Extended; Payment: Extended;
  const PresentValue, FutureValue: Extended;
  PaymentTime: TPaymentTime): Extended;
function NetPresentValue(const Rate: Extended;
  const CashFlows: array of Double;
  PaymentTime: TPaymentTime): Extended;
function Payment(Rate: Extended; NPeriods: Integer;
  const PresentValue, FutureValue: Extended;
  PaymentTime: TPaymentTime): Extended;
function PeriodPayment(const Rate: Extended; Period,
  NPeriods: Integer; const PresentValue, FutureValue: Extended;
  PaymentTime: TPaymentTime): Extended;
function PresentValue(const Rate: Extended; NPeriods: Integer;
  const Payment, FutureValue: Extended;
  PaymentTime: TPaymentTime): Extended;
function SLNDepreciation(const Cost,      //Straight-line depn.
  Salvage: Extended; Life: Integer): Extended;
function SYDDepreciation(const Cost,      //Sum of yrs depn.
  Salvage: Extended;  Life, Period: Integer): Extended;
```

These are pretty straightforward, if not necessarily as flexible as the equivalent functions in Microsoft Excel:

```
uses
  System.SysUtils, System.Math;

var
  Cost, ResaleValue: Extended;
  YearsOfUse, Year: Integer;
begin
  Cost := 2000;
  YearsOfUse := 4;
  ResaleValue :=  200;
  for Year := 1 to YearsOfUse do
    WriteLn('Year ', Year, ': £', FloatToStr(
      DoubleDecliningBalance(Cost, ResaleValue,
        YearsOfUse, Year)));
end.
```

This calculates year 1 as £1000, year 2 as £500, year 3 as £250, and year 4 as £50.

Converting from strings

String conversions for floats broadly follow the pattern for integers. For converting *from* a string, there is one set of functions (`StrToFloat`, `StrToFloatDef` and `TryStrToFloat`) overloaded for `Single`, `Double` and `Extended`, and another set (`StrToCurr`, `StrToCurrDef` and `TryStrToCurr`) that work with `Currency` values:

```
var
  CurrValue: Currency;
  DblValue: Double;
  SnlValue: Single;
begin
  CurrValue := StrToCurr(S);
  DblValue := StrToFloatDef(S, 0);
  if TryStrToFloat(S, SnlValue) then //...
```

In every case, there are overloads for passing an explicit `TFormatSettings` record. This allows you to force treating the decimal separator as (for example) a full stop for persistence purposes. Otherwise, the decimal separator defined by the computer's localisation settings will be used :

```
const
  AngloStr = '123.56';
  FrenchStr = '123,56';
var
  FrenchSettings, BritishSettings: TFormatSettings;
  V: Double;
begin
  FrenchSettings := TFormatSettings.Create('fr-FR');
  BritishSettings := TFormatSettings.Create('en-GB');
  WriteLn('Anglophone-style test string (', AngloStr, ')');
  WriteLn('Naive TryStrToFloat: ', TryStrToFloat(AngloStr, V));
  WriteLn('TryStrToFloat with fr-FR: ',
    TryStrToFloat(AngloStr, V, FrenchSettings));
  WriteLn('TryStrToFloat with en-GB: ',
    TryStrToFloat(AngloStr, V, BritishSettings));
  WriteLn('Francophone-style test string (', FrenchStr, ')');
  WriteLn('Naive TryStrToFloat: ', TryStrToFloat(FrenchStr, V));
  WriteLn('TryStrToFloat with fr-FR: ',
    TryStrToFloat(FrenchStr, V, FrenchSettings));
  WriteLn('TryStrToFloat with en-GB: ',
    TryStrToFloat(FrenchStr, V, BritishSettings));
```

To literally just force a decimal separator, call the parameterless version of `TFormatSettings.Create` and set the `DecimalSeparator` field:

```
Settings := TFormatSettings.Create;
Settings.DecimalSeparator := '.';
```

As with the integer functions, any thousand separator will cause the conversion to fail. Furthermore, and despite the type name, the currency functions do *not* accept a leading currency symbol:

```
CurrValue := StrToCurr('£2.50', BritishSettings); //exception!
```

Converting to strings

The simplest way to convert a float to a string is to call `FloatToStr` for a regular float and `CurrToStr` for a `Currency` value:

```
S := FloatToStr(SomeDbl);
S := CurrToStr(SomeCurr);
```

When the value passed to `FloatToStr` is a NaN, then `'NAN'` is returned; if positive infinity, then `'INF'`; and if negative infinity, then `'-INF'`. Otherwise, the so-called 'general' number format is used. As with all float formatting routines, `FloatToStr` and `CurrToStr` are overloaded with versions that take an explicit `TFormatSettings` record. In this case, that just allows fixing the decimal separator, as it did for `StrToFloat` and friends.

For slightly more control, `FloatToStrF` and `CurrToStrF` may be used:

```
type
  TFloatFormat = (ffGeneral, ffExponent, ffFixed, ffNumber, ffCurrency);

function FloatToStrF(Value: Extended; Format: TFloatFormat;
  Precision, Digits: Integer): string;
function FloatToStrF(Value: Extended; Format: TFloatFormat;
  Precision, Digits: Integer; const Settings: TFormatSettings): string;

function CurrToStrF(Value: Currency; Format: TFloatFormat;
  Digits: Integer): string;
function CurrToStrF(Value: Currency; Format: TFloatFormat;
  Digits: Integer; const Settings: TFormatSettings): string;
```

In the case of `FloatToStrF`, `Precision` specifies the total number of digits before the exponent (i.e., the `E`) when using scientific format. This should be between 2 and 18 when using the 32 bit compiler, and between 2 and 16 when using the 64 bit compiler. The `Format` parameter works as thus:

- `ffGeneral` returns the shortest string representation possible, stripping off trailing zeros as a result. The `Digits` parameter is ignored, and normal rather than scientific format will be preferred so long as `Precision` is big enough:

```
  S := FloatToStrF(123456.70, ffGeneral, 18, 3);  //123456.7
  S := FloatToStrF(123456.70, ffGeneral, 2, 3);   //1.2E005
```

- `ffExponent` forces scientific format to be used, with `Digits` specifying the minimum number of digits in the exponent (i.e., after the `E`):

```
  //first line assigns 1.23456700000000000E+005, second 1.2E+005
  S := FloatToStrF(123456.70, ffExponent, 18, 3);
  S := FloatToStrF(123456.70, ffExponent, 2, 3);
```

- `ffFixed` is like `ffGeneral` , only with the `Digits` parameter being operative:

```
  S := FloatToStrF(123456.70, ffFixed, 18, 3);  //123456.700
  S := FloatToStrF(123456.70, ffFixed, 2, 3);   //1.2E005
```

- `ffNumber` is like `ffFixed`, only with thousand separators being used:

```
  S := FloatToStrF(123456.70, ffNumber, 18, 3); //123,456.700
  S := FloatToStrF(123456.70, ffNumber, 2, 3);  //1.2E005
```

  You can customise the separator by using an overload that takes an explicit `TFormatSettings` and assigning the record's `ThousandSeparator` field:

```
  var
    Settings: TFormatSettings;
    S: string;
  begin
    { Use a space for the thousand separator and a dash for the
      decimal point }
    Settings := TFormatSettings.Create;
    Settings.ThousandSeparator := ' ';
    Settings.DecimalSeparator := '-';
    { Next line assigns 123 456-700 }
    S := FloatToStrF(123456.70, ffNumber, 18, 3, Settings);
```

- `ffCurrency` formats using the system localisation settings for monetary values, taking care of both the currency symbol and its position. `Digits` specifies the number of digits after the decimal point (or equivalent):

```
  { Use system settings - will result in '£2.50' for me (UK) }
  S := FloatToStrF(2.5, ffCurrency, 18, 2);
  { Use French settings - will result in '2,50 €' this time }
  Settings := TFormatSettings.Create('fr-FR');
  S := FloatToStrF(2.5, ffCurrency, 18, 2, Settings);
```

One step beyond `FloatToStrF` and `CurrToStrF` are `FormatFloat` and `FormatCurr`:

```
function FormatFloat(const Format: string; Value: Extended): string;
function FormatFloat(const Format: string; Value: Extended;
  const AFormatSettings: TFormatSettings): string; overload;
function FormatCurr(const Format: string; Value: Currency): string;
function FormatCurr(const Format: string; Value: Currency;
  const AFormatSettings: TFormatSettings): string; overload;
```

The format string here has up to three parts separated by semi-colons: the first section specifies the format for positive values, the second for negative values and the third for zeros. If only two parts are given, then the first applies to both positive values and zeros, and if only one part, all values. If a part is left empty, then the 'general' format is used:

```
S := FormatFloat(';;"nowt"', 0);     //nowt
S := FormatFloat(';;"nowt"', 54.00); //54
```

Text wrapped by double or single quotes are outputted 'as is', as shown above. Otherwise, use `0` to force a digit, `#` to include a digit if there is one to include, a full stop/period to output the decimal separator, a comma to output the thousand separator, and one of `E+`, `E-`, `e+` or `e-` to output using scientific notation:

```
var
  FrenchSettings: TFormatSettings;
  S: string;
begin
  S := FormatFloat('000.00', 5);          //005.00 (UK locale)
  S := FormatFloat('###.##', 5);          //5
  FrenchSettings := TFormatSettings.Create('fr-FR');
  S := FormatFloat('0.0', 5, FrenchSettings); //5,0
```

If `e-` is used, a sign is only outputted when it is negative; if `e+`, when it is negative *or* positive:

```
S := FormatFloat('#####e+', 345276.1);    //34528e+1
S := FormatFloat('#####e-', 345276.1);    //34528e1
S := FormatFloat('#######e-', 345276.1);  //3452761e-1
```

Lastly, the generic `Format` function can also be used to convert a float to a string, similar to how it can be used convert an integer to a string. The relevant letter codes in the floating point case are **e**, **f**, **g**, **n** and **m**:

- **e** (exponent) outputs using scientific notation; any precision specifier (added by including a full stop and a number immediately before the letter code) sets the number of digits *before* the `E`:

```
S := Format('It is %.3e', [6148.9]); //6.15E+003
S := Format('It is %.6e', [6148.9]); //6.14890E+003
S := Format('It is %e', [6148.9]);   //6.14890000000000E+003
```

- **f** (fixed) uses the precision specifier to set the number of digits after the decimal point; the default is 2:

```
S := Format('Now it is %.3f', [6148.9]); //6148.900
S := Format('Now it is %.6f', [6148.9]); //6148.900000
S := Format('Now it is %f', [6148.9]);   //6148.90
```

- **n** (number) acts as **f**, but with the addition of thousand separators:

```
S := Format('%.3n you say?', [6148.9]); //6,148.900
S := Format('%.6n you say?', [6148.9]); //6,148.900000
S := Format('%n you say?', [6148.9]);   //6,148.90
```

- **g** (general) outputs the shortest possible string; any precision specifier controls the number of significant digits (the default is 15):

```
S := Format('Yes, %.3g', [6148.9]); //Yes, 6.15E003
S := Format('Yes, %.6g', [6148.9]); //Yes, 6148.9
S := Format('Yes, %g', [6148.9]);   //Yes, 6148.9
```

- **m** (money) outputs using the system localisation settings for monetary values. Any precision specifier controls the number of digits after the decimal point; if no specifier is given, the locale default is used (usually 2 for a Western country):

```
S := Format('It cost %.3m', [6148.9]); //£6,148.900
S := Format('It cost %.6m', [6148.9]); //£6,148.900000
S := Format('It cost %m', [6148.9]);   //£6,148.90
```

  When an explicit `TFormatSettings` record is passed, the default number of decimal places will be the value of its `CurrencyDecimals` field:

```
uses System.SysUtils;

var
  Settings: TFormatSettings;
begin
  //use system default; outputes £999.00 for me
  WriteLn(Format('%m', [999.0]));
  //force only one digit after the decimal place (£999.0)
  Settings := TFormatSettings.Create;
  Settings.CurrencyDecimals := 1;
  WriteLn(Format('%m', [999.0], Settings));
end.
```

Working with dates, times and timings

The TDateTime time

Delphi's standard type for storing dates and times, `TDateTime`, is a floating point type — specifically, a `Double` — in which the (signed) integral part denotes the number of days since 30th December 1899 and the (unsigned) fractional part the time of day. By design, `TDateTime` is the same thing as the date/time type used by OLE Automation on Windows (`VT_DATE`, alias the `Date` type in the Visual Basic for Applications), however Delphi's date/time support isn't dependent on OLE itself.

By convention, a `TDateTime` value is usually considered to lie within the local time zone. That doesn't *have* to be the case however. In practice, if the time zone matters, you need to know what what one a given `TDateTime` is supposed to lie in, since the value won't keep track of that itself.

Given its basic format, in principle, finding the number of days between two `TDateTime` values is very simple — simply subtract the earlier from the later date and truncate:

```
function NumOfDaysBetween(const ANow, AThen: TDateTime): Integer;
begin
  Result := Trunc(ANow - AThen);
end;
```

For historical reasons, the precise semantics of OLE date/time values around 0.0, and therefore, Delphi `TDateTime` values around that same number, get a bit tricky though. As a result, if you may be handling dates around the year 1900, you should consider using the RTL's helper functions instead of simple subtraction operations.

These functions are all to be found in the `System.DateUtils` unit with names such as `DaysBetween`, `YearsBetween`, `MonthsBetween`, `WeeksBetween`, `HoursBetween`, `MinutesBetween`, `SecondsBetween` and `MilliSecondsBetween`. Each of these also has a `xxxSpan` variant (e.g. `DaysSpan`, `YearsSpan` etc.); where the `xxxBetween` versions return whole units, rounding down if necessary, the `xxxSpan` variants return a `Double` that includes the fractional number of units too:

```
uses
  System.SysUtils, System.DateUtils;

var
  Date1, Date2: TDateTime;
  WholeMonths: Integer;
  Months: Double;
begin
  Date1 := StrToDate('10/10/2012');
  Date2 := StrToDate('01/01/2011');
  WholeMonths := MonthsBetween(Date1, Date2); //21
  Months := MonthSpan(Date1, Date2);          //21.2895277207392
```

Useful constants for units of time

The `System.SysUtils` unit contains a number of useful constants concerning units of time:

```
const
  HoursPerDay   = 24;
  MinsPerHour   = 60;
  SecsPerMin    = 60;
  MSecsPerSec   = 1000;
  MinsPerDay    = HoursPerDay * MinsPerHour;
  SecsPerDay    = MinsPerDay * SecsPerMin;
  SecsPerHour   = SecsPerMin * MinsPerHour;
  MSecsPerDay   = SecsPerDay * MSecsPerSec;
```

`System.DateUtils` provides a few more:

```
const
  DaysPerWeek = 7;
  WeeksPerFortnight = 2;
  MonthsPerYear = 12;
  YearsPerDecade = 10;
  YearsPerCentury = 100;
  YearsPerMillennium = 1000;
  OneHour = 1 / HoursPerDay;
  OneMinute = 1 / MinsPerDay;
  OneSecond = 1 / SecsPerDay;
  OneMillisecond = 1 / MSecsPerDay;
  ApproxDaysPerMonth: Double = 30.4375; //over a 4 year span
  ApproxDaysPerYear: Double  = 365.25;  //ditto
  DaysPerYear: array [Boolean] of Word = (365, 366);
```

In case of `DaysPerYear`, you still need to know whether the year in question is a leap year or not. For that, you can call the `IsLeapYear` function:

```
if IsLeapYear(2000) then WriteLn('2000 was a leap year');
if not IsLeapYear(1900) then WriteLn('1900 was NOT a leap year');
```

Getting and amending TDateTime values (Now, Date, Time, EncodeXXX, XXXOf etc.)

To return the current date and time, call the `Now` function from `System.SysUtils`; alternatively, call `Time` (or `GetTime`) to return just the current time, and `Date` to return just the current date. All three functions return values in the local time zone.

The various component parts of a date from a human point of view (year, month, day, hours, minutes, seconds and milliseconds) can be extracted from a `TDateTime` value using the `DecodeDate` and `DecodeTime` procedures of `SysUtils`, or alternatively, a number of routines in `System.DateUtils`: `DecodeDateTime` extracts everything, and a series of `xxxOf` functions (e.g. `YearOf`, `MonthOf`) extract individual parts. In all cases, the month is returned as a number whereby January = 1, February = 2 etc., and the day is the day of the month rather than the day of the week:

```
var
  Year, Month, Day, Hour, Minute, Second, MSecond: Word;
  Value: TDateTime;
begin
  DecodeDateTime(Now, Year, Month, Day, Hour, Minute, Second, MSecond);
  WriteLn('The current year is ', Year);
  WriteLn('The current month is no. ', Month, ' of 12');
  WriteLn('The current day of the month is ', Day);
  WriteLn('The current time is ', Hour, ':', Minute);
  WriteLn('Yesterday was in ',
    FormatSettings.LongMonthNames[MonthOf(Now - 1)]);
```

The `xxxOf` functions also have sibling routines to extract the hour or whatever relative to the start of the year (e.g. `HourOfTheYear`), the month (e.g. `MinuteOfTheMonth`), the week (e.g. `HourOfTheWeek`), the day (e.g. `MinuteOfTheDay`), the hour (for minutes, seconds and milliseconds), and the minute (for seconds and milliseconds). For completeness' sake, there is also a `MillisecondOfTheSecond` function, though by definition, this does exactly the same thing as `MillisecondOf`.

To construct a `TDateTime` value in the first place, you can call either `EncodeDate` or `EncodeTime` from `SysUtils`, or `EncodeDateTime` from `DateUtils`. For example, the following constructs a `TDateTime` set to ten minutes past midday on 30th November 2011 (both seconds and milliseconds are left at 0):

```
MyDT := EncodeDateTime(2011, 11, 30, 12, 10, 0, 0);
```

`System.DateUtils` also provides the following variants:

- `EncodeDateDay` returns a `TDateTime` value for the *n*th day of the specified year, where 1st January is day 1 and 31st December is either day 365 or (if a leap year) day 366. `EncodeDateDay(2011, 364)` therefore returns a value for 30th December 2011.
- `EncodeDateMonthWeek` takes arguments for the year, month, week of the month and day of week, where 1 = Monday, 2 = Tuesday and so on. The first week of a month is defined as the first Mon-Sun cycle with four or more days. Given 1st January 2011 was a Saturday, `EncodeDateMonthWeek(2011, 1, 1, 7)` therefore returns a value for 9th January.

  To make things a bit clearer, constants are predefined for both the months and days of the week: the previous example may therefore be written as `EncodeDateMonthWeek(2011, MonthJanuary, 1, DaySunday)`.
- `EncodeDateWeek` returns the value for a given day of a given week of a given year. Once more, week 1 is defined as the first calendar week with four or more days.
- `EncodeDayOfWeekInMonth` outputs a value for a given *n*th day of the week for a given month for a given year. `EncodeDayOfWeekInMonth(2011, MonthApril, 2, DayFriday)` therefore retrieves a `TDateTime` value for the second Friday of April 2011, i.e. 8th April 2011.

If any argument passed to one of the `EncodeXXX` functions is invalid (e.g., you specify 13 for the month), an `EConvertError` exception is raised. If there is a reasonable chance of this happing, you may instead prefer to call the function's `TryEncodeXXX` equivalent. Like `TryStrToInt`, a `TryEncodeXXX` function returns `False` on invalid input rather than raising an exception:

```
var
  DT: TDateTime;
begin
  if TryEncodeDate(2011, 07, 06, DT) then //6th July: fine
    Write('OK')
  else
    Write('not OK');
  if TryEncodeDate(2010, 08, 32, DT) then //32nd Aug: bad!
    Write(', OK')
  else
    Write(', not OK');
  //output: OK, not OK
end.
```

`System.DateUtils` also provides various routines for amending a `TDateTime` value: `RecodeDate` (for all elements of the date part), `RecordTime` (for all elements of the time part), `RecodeDateTime` (for all of it), `RecodeYear`, `RecodeMonth`, `RecodeDay`, `RecodeHour`, `RecodeMinute`, `RecodeSecond`, and `RecodeMillisecond`. These are all functions that return the revised date/time as their result, leaving the parameter unchanged:

```
var
  DT1, DT2: TDateTime;
begin
  DT1 := Now;
  DT2 := RecodeYear(DT1, YearOf(DT1) - 1);
  WriteLn('Today is ', DateToStr(DT1));
  WriteLn('A year ago today was ', DateToStr(DT2));
```

Also available in `System.DateUtils` are `IncYear`, `IncMonth`, `IncWeek`, `IncDay`, `IncHour`, `IncMinute`, `IncSecond` and `IncMilliSecond` functions (all but `IncMonth` are declared in `System.DateUtils`; `IncMonth` is in `System.SysUtils`). These increment the relevant part of a date/time value by 1, or if a second argument is passed, a given whole number (which can be negative). `IncWeek(Date, -2)` therefore returns a value for a fortnight ago.

Converting TDateTime values to strings

The simplest way to convert either the date portion of a `TDateTime` value to a string, the time portion to a string, or both the date and the time portion to a string, is to call `DateToStr`, `TimeToStr` or `DateTimeToStr` as appropriate (all are declared by `System.SysUtils`). By default, `DateToStr` uses the user's 'short' date format (e.g. 6/6/2011) and `TimeToStr` the user's 'long' time format (e.g. 12:18:05 for eighteen minutes and five seconds past midday).

In themselves, `DateToStr` and `TimeToStr` offer no direct control over their output however. If control is what you require, use either `DateTimeToString` (a procedure) or `FormatDateTime` (a function) instead:

```
procedure DateTimeToString(var Result: string;
  const Format: string; DateTime: TDateTime);
function FormatDateTime(const Format: string;
  DateTime: TDateTime): string;
```

The `Format` parameter is a string whose parts are expanded as thus, using localisation settings from the operating system:

- `c` expands to the date part of the 'short date' format, followed by a space and the time part of the 'long time' format. In the United Kingdom, this produces a string such as `'08/04/2012 12:10:00'` for ten minutes past midday on 8th April 2012. Using `FormatDateTime('c', DT)` is the same as calling `DateTimeToStr`.
- `d` expands to the day of the month without a leading zero. Thus, if the date is 8th April, then `d` will expand to 8.
- `dd` expands to the day of the month with a leading zero if it is less than 10. Passing a date for the 8th of a month will therefore output 08; for the 11th, 11.
- `ddd` expands to the day of the week as a short abbreviation, e.g. Sun or Mon.
- `dddd` expands to the full name of the day of the week, e.g. Sunday or Monday.
- `ddddd` expands to the whole date in the 'short date' format, e.g. 08/04/2012.
- `dddddd` expands to the whole date in the 'long date' format, e.g. 08 April 2012.
- `m` expands to the month of the year without a leading zero, e.g. a date in April will have `m` expand to 4.
- `mm` expands to the month of the year with a leading zero if necessary, e.g. a date in April will output 04.
- `mmm` expands to the month as a short abbreviation, e.g. Apr for April.
- `mmmm` expands to the full name of the month, e.g. April.
- `yy` outputs the year as a two digit number, e.g. 12 for a date in 2012.
- `yyyy` outputs the year as a four digit number, e.g. 2012.
- `h` outputs the hour without any leading zero, e.g. 8.

- `hh` outputs the hour with a leading zero if necessary, e.g. 08.
- `am/pm` outputs am or pm as appropriate, and modifies any included `h` or `hh` specifier to use the 12 hour format. If you use `AM/PM`, or `Am/Pm`, the casing will be respected:

```
  var
    DT: TDateTime;
    S: string;
  begin
    DT := EncodeTime(13, 10, 4, 0);
    S := FormatDateTime('h:m am/pm', DT);   //1:10 pm
    S := FormatDateTime('(AM/PM) h.m', DT); //(PM) 1.10
```

- `a/p` outputs a or p as appropriate, and modifies any included `h` or `hh` specifier to use the 12 hour format. If you use `A/P`, the casing will be respected.
- `ampm` outputs am or pm as localised at the operating system level, and modifies any included `h` or `hh` specifier to use the 12 hour format.
- `n` outputs the minute of the hour without a leading zero.
- `nn` outputs the minute of the hour with a leading zero if necessary.
- `s` outputs the second of the minute without a leading zero.
- `ss` outputs the second of the minute with a leading zero if necessary.
- `z` outputs the millisecond of the second without a leading zero.
- `zzz` outputs the millisecond of the second with one or two leading zero if necessary.
- `t` outputs the complete time in the operating system-defined ‘short time’ format.
- `tt` outputs the complete time in the operating system-defined ‘long time’ format.
- `/` outputs the operating system-defined date separator character. In an Anglophone context this will usually be a forward slash.
- `:` outputs the operating system-defined time separator character. In an Anglophone context this will usually be a colon.

To display any of the listed special sub-strings ‘as is’, wrap them in either single or double quotes. In fact, the safest way to deal with ordinary text outside of space characters is to enclose it *all* in quotes:

```
S := FormatDateTime('"The time now:" h.mm am/pm', Time);
```

Converting from a string to a TDateTime (StrToXXX, TryStrToXXX)

Functions are available too to parse a date string, a time string, or a date-and-time string:

```
function StrToDate(const S: string): TDateTime;
function StrToDateDef(const S: string;
  const Default: TDateTime): TDateTime;
function TryStrToDate(const S: string; out Value: TDateTime): Boolean;
function StrToTime(const S: string): TDateTime;
function StrToTimeDef(const S: string;
  const Default: TDateTime): TDateTime;
function TryStrToTime(const S: string;
  out Value: TDateTime): Boolean;
function StrToDateTime(const S: string): TDateTime;
function StrToDateTimeDef(const S: string;
  const Default: TDateTime): TDateTime;
function TryStrToDateTime(const S: string;
  out Value: TDateTime): Boolean;
```

When the string cannot be parsed, `TryStrToXXX` returns `False`, `StrToXXXDef` returns the specified default value, and `StrToXXX` raises an `EConvertError` exception.

The date and date/time versions are limited by not understanding month names, with the consequence that something like `StrToDate('26 March 2012')` will raise an exception. Parsing is also tightly bound to current localisation settings. Since my computer uses / as the date separator, the following will therefore raise an exception:

```
  DT := StrToDate('01-01-2011');
```

If you wish to be more flexible, you will have to code the flexibility yourself:

```
function TryStrToDateNice(S: string; out DT: TDateTime): Boolean;
var
```

```
  I: Integer;
begin
  for I := 1 to Length(S) do
    case S[I] of
      '\', '-', '/': S[I] := FormatSettings.DateSeparator;
    end;
  Result := TryStrToDate(S, DT);
end;
```

When interpreting date strings with two digit years (e.g. 31/06/11), a fifty year 'pivot' is used by default. Thus, if the current year is 2011, then a date string of 1/1/60 is interpreted as 1st January 2060 and a string of 1/1/61 is interpreted as 1st January 1961.

Using customised localisation settings (TFormatSettings record)

Similar to the integer and float routines we met earlier, all the functions for converting to and from strings and `TDateTime` values can make use of a `TFormatSettings` record. If one isn't passed explicitly, then the global `FormatSettings` instance is used, which is initialised with the settings of the system locale. Being a simple global variable, you can access the fields of `FormatSettings` as you wish:

```
uses System.SysUtils;

var
  DayOfWeek: string;
begin
  Write('The days of the week in the current locale are');
  for DayOfWeek in FormatSettings.LongDayNames do
    Write(' ', DayOfWeek);
  { My output: The days of the week in the current locale are
    Sunday Monday Tuesday Wednesday Thursday Friday Saturday}
end.
```

In all, a `TFormatSettings` record has the following public fields:

```
CurrencyString: string;
CurrencyFormat, CurrencyDecimals: Byte;
DateSeparator, TimeSeparator, ListSeparator: Char;
ShortDateFormat, LongDateFormat: string;
TimeAMString, TimePMString: string;
ShortTimeFormat, LongTimeFormat: string;
ShortMonthNames, LongMonthNames: array[1..12] of string;
ShortDayNames, LongDayNames: array[1..7] of string;
ThousandSeparator, DecimalSeparator: Char;
TwoDigitYearCenturyWindow: Word; //defines the 'pivot'
NegCurrFormat: Byte;
```

While you can alter any of the `FormatSettings` fields directly, it is better to create your own `TFormatSettings` instance. This is done via one of three `Create` functions: a parameterless version that loads the current OS localisation settings, a version that takes a LCID on Windows or a `CFLocaleRef` on OS X, and a version that takes a locale string such as `en-gb`:

```
uses
  System.SysUtils,
  {$IFDEF MSWINDOWS}Winapi.Windows{$ENDIF}
  {$IFDEF MACOS}Macapi.CoreFoundation{$ENDIF};

var
  Sys, American, Finnish: TFormatSettings;
  {$IFDEF MACOS}
  LocaleRef: CFLocaleRef;
  {$ENDIF}
begin
  Sys := TFormatSettings.Create;
  American := TFormatSettings.Create('en-us');
  {$IFDEF MSWINDOWS}
  Finnish := TFormatSettings.Create(LANG_FINNISH);
  {$ENDIF}
  {$IFDEF MACOS}
  LocaleRef := CFLocaleCreate(nil, CFSTR(UTF8String('fi')));
  try
    Finnish := TFormatSettings.Create(LocaleRef);
  finally
    CFRelease(LocaleRef);
  end;
  {$ENDIF};
```

If you need the settings for a specific locale, you should generally use the string version — `TFormatSettings.Create('fi')` not `TFormatSettings.Create(LANG_FINNISH)`. Aside from being OS X compatible, modern versions of Windows only support LCIDs for reasons of backwards compatibility.

In the following example, first a variant of the system settings are used, then second the default Dutch settings are loaded:

```
const
  DutchMsg = '"Vandaag is het" dddd "en de maand is" mmmm';
var
  Settings: TFormatSettings;
  S: string;
begin
  Settings := TFormatSettings.Create;
  Settings.ShortDateFormat := 'yyyy/mm/dd';
  WriteLn('Today''s date is ' + DateToStr(Date, Settings));

  Settings := TFormatSettings.Create('nl-NL');
  S := FormatDateTime(Msg, Date, Settings);
```

When persisting date/times as human-readable strings, it is a good idea to *always* use an explicit `TFormatSettings` record, hard coding relevant fields. Doing so will avoid ambiguity when reading back in previously written data. For example, does 04/05/2011 mean the fourth of May or the 5th of April?

Converting to and from other date/time formats

The majority of Delphi's standard libraries assume date/time values are encoded as instances of `TDateTime`. Since there are many other possible ways to encode date/times though, the RTL provides some conversion functions for the more common ones:

- `DateTimeToUnix` and `UnixToDateTime` (`System.DateUtils`) convert to and from Unix `time_t` values. This is useful when working with the POSIX API on OS X, or perhaps a file structure that uses POSIX date/time values:

```
uses System.SysUtils, System.DateUtils, Posix.SysTypes, Posix.Time;

var
  DelphiDT: TDateTime;
  UnixDT: time_t;
  S: string;
begin
  DelphiDT := Now;     //returns in local time zone
  UnixDT := time(nil); //returns in UTC/GMT
  WriteLn('The Delphi RTL reckons it is ',
    DateTimeToStr(DelphiDT));
  WriteLn('The POSIX API reckons it is ',
    DateTimeToStr(UnixToDateTime(UnixDT)));
end.
```

- On Windows, `DateTimeToFileDate` and `FileDateToDateTime` (`System.SysUtils`) convert to and from MS-DOS file times, as packed into an `Integer`/`Int32`.
- Also on Windows, `DateTimeToSystemTime` and `SystemTimeToDateTime` (`System.SysUtils`) convert to and from the Windows API `SYSTEMTIME` struct (alias the `TSystemTime` record):

```
uses System.SysUtils, Winapi.Windows;

var
  DelphiDT: TDateTime;
  SysDT: TSystemTime;
  S: string;
begin
  DelphiDT := Now;     //returns in local time zone
  GetLocalTime(SysDT); //also returns in local time zone
  WriteLn('The Delphi RTL reckons it is ',
    DateTimeToStr(DelphiDT));
  WriteLn('The Windows API reckons it is ',
    DateTimeToStr(SystemTimeToDateTime(SysDT)));
end.
```

- `DateTimeToXMLTime` and `XMLTimeToDateTime` (`Soap.XSBuiltIns` unit) convert to and from XML date/time strings. Since these can carry time zone information, the functions take an optional `Boolean` parameter. In the case of `DateTimeToXMLTime`, it determines whether the source `TDateTime` is treated as being in the local time zone rather than UTC/GMT, the default being `True` (i.e., *do* interpret it as a local time):

```
uses System.SysUtils, System.DateUtils, Soap.XSBuiltIns;
```

```
  var
    DT: TDateTime;
    S: string;
  begin
    { Assign 8 April 2012, 10.15 am and 30.5s }
    DT := EncodeDateTime(2012, 4, 8, 10, 15, 30, 500);
    { As I am in British Summer Time, which is 1 hour ahead
      of GMT/UTC, the following will assign
      '2012-04-08T10:15:30.500+01:00' to S }
    S := DateTimeToXMLTime(DT);
    { The next line will assign '2012-04-08T10:15:30.500Z', which
      is to understand the TDateTime source as being GMT/UTC }
    S := DateTimeToXMLTime(DT, False);
```

In the case of `XMLTimeToDateTime`, the optional parameter determines whether the `TDateTime` returned should be in the local time zone or not (the default is `False`, which means not returning a *universal* time). In the string input, either the time section in its entirety or just its final parts are optional; nothing after the Z means GMT/UTC:

```
  var
    DT: TDateTime;
    S: string;
  begin
    DT := XMLTimeToDateTime('1973-03-23T09:45Z');
    { Again, being 1 hour ahead of GMT/UTC will mean the
      next line outputs '23/03/1973 10:45:00' }
    WriteLn(DateTimeToStr(DT));
    { Now ignore the time zone }
    DT := XMLTimeToDateTime('1973-03-23T09:45Z', True);
    { Outputs '23/03/1973 09:45:00' }
    WriteLn(DateTimeToStr(DT));
```

Lastly, the RTL internally uses a `TTimeStamp` type for much of its date/time twiddling:

```
type
  TTimeStamp = record
    Time: Integer;
    Date: Integer;
  end;
```

Here, `Time` is the number of milliseconds since midnight, and `Date` is the number of days plus one since 1st January, 1ad. Such a structure provides greater accuracy for the time part than a `TDateTime`, so you may occasionally wish to use `TTimeStamp` yourself. The names of functions to convert to and from `TDateTime` are as you would expect — `DateTimeToTimeStamp` and `TimeStampToDateTime`:

```
uses System.SysUtils, System.DateUtils;

var
  DT: TDateTime;
  TS: TTimeStamp;
begin
  { assign the value for 9 May 2012 12:30:15 }
  DT := EncodeDateTime(2012, 05, 09, 12, 30, 15, 0);
  TS := DateTimeToTimeStamp(DT);
  Write('Days since 1st January, 1AD: ');
  WriteLn(TS.Date - 1);                 //output: 734631
  Write('Milliseconds since midnight: ');
  WriteLn(TS.Time);                     //output: 45015000
end.
```

Working with the local time zone and UTC/GMT (TTimeZone)

Declared in `System.DateUtils`, the `TTimeZone` class does three things: provide information about the local time zone (e.g. its UTC/GMT offset and name), convert date/time values between the local time zone and UTC/GMT, and verify what 'type' a given local time is, for example whether it occurs during a period of daylight savings time.

Being an abstract class, you don't explicitly instantiate `TTimeZone`. Instead, you work with its `Local` class property. This returns an instance of a private `TTimeZone` descendant that implements all the various abstract methods of the parent class by polling the operating system for the necessary information:

```
if TTimeZone.Local.IsDaylightTime(Now) then
  WriteLn('We are currently in daylight savings time')
else
  WriteLn('We are NOT currently in daylight savings time');
```

The public interface of `TTimeZone.Local` is composed of four properties and several methods. Of the properties, `ID` returns the name of the current time zone, `DisplayName` the name of current seasonal time zone, `UTCOffset` the current offset from Coordinated Universal Time (which just means GMT if you're from the UK), and `Abbreviation` a short abbreviated name for the current time zone:

```
WriteLn('The current time zone ID is ', TTimeZone.Local.ID);
WriteLn('The current time zone name is ', TTimeZone.Local.DisplayName);
WriteLn('The current time zone abbreviation is ',
  TTimeZone.Local.Abbreviation);
WriteLn('The current time zone''s offset from GMT is ',
  FloatToStr(TTimeZone.Local.UtcOffset.TotalHours), ' hour(s)');
```

The `TTimeZone` methods fall into three groups. The first correspond to the properties just mentioned, only taking a specific date/time as a parameter instead of assuming the current date and time:

```
function GetAbbreviation(const ADateTime: TDateTime;
  const ForceDaylight: Boolean = False): string;
function GetDisplayName(const ADateTime: TDateTime;
  const ForceDaylight: Boolean = False): string;
function GetUtcOffset(const ADateTime: TDateTime;
  const ForceDaylight: Boolean = False): TTimeSpan;
```

The optional `ForceDaylight` parameter is used in the case of a time that could be in either standard or daylight saving time. For example, when the clocks go back, there will be a period (typically one hour) in which clock times happen twice.

The second group of methods return basic information about a given date/time:

```
type
  TLocalTimeType = (lttStandard, lttDaylight, lttAmbiguous, lttInvalid);

function GetLocalTimeType(const ADateTime: TDateTime): TLocalTimeType;
function IsAmbiguousTime(const ADateTime: TDateTime): Boolean;
function IsDaylightTime(const ADateTime: TDateTime;
  const ForceDaylight: Boolean = False): Boolean;
function IsInvalidTime(const ADateTime: TDateTime): Boolean;
function IsStandardTime(const ADateTime: TDateTime;
  const ForceDaylight: Boolean = False): Boolean;
```

Here, an 'ambiguous' time is one that falls in a period when the clocks go back, and an 'invalid' time one that falls in a period when the clocks go forward. For example, if on a particular date the clocks go forward one hour at 1:00, times from 1:00-1:59 will be invalid.

The final pair or methods convert between local and 'universal' (i.e., UTC/GMT) time. E.g., during the summer months, midday in Berlin would be 10 am in Greenwich Mean Time:

```
function ToLocalTime(const ADateTime: TDateTime): TDateTime;
function ToUniversalTime(const ADateTime: TDateTime;
  const ForceDaylight: Boolean = False): TDateTime;
```

In use:

```
var
  DT: TDateTime;
begin
  DT := TTimeZone.Local.ToUniversalTime(Now);
  WriteLn('The current time in UTC is ', TimeToStr(DT));
```

Working with time periods (TTimeSpan)

The `TTimeSpan` type is for when you need to work with time *durations* rather than specific points in time. Declared in the `System.TimeSpan` unit, `TTimeSpan` is a record type whose instances hold a signed 'ticks' value (each tick is 1/10,000 of a millisecond). This value can then be read expressed in days, hours, minutes, seconds and milliseconds:

```
property Ticks: Int64 read FTicks;
property Days: Integer read GetDays;
property Hours: Integer read GetHours;
property Minutes: Integer read GetMinutes;
property Seconds: Integer read GetSeconds;
property Milliseconds: Integer read GetMilliseconds;
property TotalDays: Double read GetTotalDays;
property TotalHours: Double read GetTotalHours;
property TotalMinutes: Double read GetTotalMinutes;
property TotalSeconds: Double read GetTotalSeconds;
property TotalMilliseconds: Double read GetTotalMilliseconds;
```

Given a ticks value for a day and a half exactly, the properties of a `TTimeSpan` object will read as thus:

- `Ticks` will report 1,296,000,000,000 (10,000 x 1,000 x 60 x 60 x 24 x 1.5).
- `Days` will report 1.
- `Hours` will report 12.
- `Minutes`, `Seconds` and `Milliseconds` will all report 0.
- `TotalDays` will report 1.5.
- `TotalHours` will report 36.
- `TotalMinutes` will report 2,160 (60 x 24 x 1.5).
- `TotalSeconds` will report 129,600 (60 x 60 x 24 x 1.5).
- `TotalMilliseconds` will report 129,600,000 (1,000 x 60 x 60 x 24 x 1.5).

Constructing a TTimeSpan object

The `TTimeSpan` type defines numerous constructors and constructing class methods for itself:

```
constructor Create(ATicks: Int64);
constructor Create(Hours, Minutes, Seconds: Integer);
constructor Create(Days, Hours, Minutes, Seconds: Integer);
constructor Create(Days, Hours, Minutes, Secs, MSecs: Integer);

class function FromDays(Value: Double): TTimeSpan; static;
class function FromHours(Value: Double): TTimeSpan; static;
class function FromMinutes(Value: Double): TTimeSpan; static;
class function FromSeconds(Value: Double): TTimeSpan; static;
class function FromMilliseconds(Value: Double): TTimeSpan; static;
class function FromTicks(Value: Int64): TTimeSpan; static;
```

In use:

```
uses
  System.SysUtils, System.TimeSpan;

var
  Duration: TTimeSpan;
begin
  Duration := TTimeSpan.FromDays(1.5);        //day and a half
  WriteLn(FloatToStr(Duration.TotalMinutes)); //2160
  Duration := TTimeSpan.Create(5, 9, 45);     //5hrs 9mins 45secs
  WriteLn(FloatToStr(Duration.TotalHours));   //5.1625
```

`Parse` and `TryParse` are also available to construct from a string:

```
class function Parse(const S: string): TTimeSpan; static;
class function TryParse(const S: string;
  out Value: TTimeSpan): Boolean;
```

Strings must have the following format, whereby `d` is the days, `h` the hours, `m` the minutes and `s` the seconds:

```
d.h:m:s
```

Any of the following abbreviated versions are acceptable too:

```
d.h:m
d.h
d
h:m:s
```

The string may also begin with a minus sign to denote a negative time span:

```
TS := TTimeSpan.Parse('6');       //6 days exactly
TS := TTimeSpan.Parse('1.10:2');  //1 day, 10 hours and 2 minutes
TS := TTimeSpan.Parse('0:3:3.4'); //3 minutes and 3.4 seconds
TS := TTimeSpan.Parse('-4:1');    //minus 4 hours and 1 minute
```

One small potential ‘gotcha’ is that both `Parse` and `TryParse` are hardcoded to work with full stops and colons only, regardless of whether the local computer has been localised to use something different. Use something other than a full stop and comma, and `Parse` will raise an exception and `TryParse` return `False`.

TTimeSpan manipulation

Since a `TTimeSpan` object can have a negative value, a `Duration` property is available to return the held value without a sign, being in effect the `TTimeSpan` equivalent of the `Abs` standard function for ordinary integers:

```
var
  TS1, TS2: TTimeSpan;
begin
  TS1 := TTimeSpan.Create(-40000);
  TS2 := TTimeSpan.Create(40000);
  WriteLn('TS1.Ticks = ', TS1.Ticks, ',
    TS1.Duration.Ticks = ', TS1.Duration.Ticks);
  WriteLn('TS2.Ticks = ', TS2.Ticks, ',
    TS2.Duration.Ticks = ', TS2.Duration.Ticks);
end.
```

This outputs the following:

```
TS1.Ticks = -40000, TS1.Duration.Ticks = 40000
TS2.Ticks = 40000, TS2.Duration.Ticks = 40000
```

Via a judicious use of operator overloading, `TTimeSpan` also supports addition, subtraction, and comparison testing: specifically, the +, -, =, <>, >, >=, <, <= and := operators can be used with `TTimeSpan` objects. The assignment operator is also overloaded to return a string representation of the object, using the same basic format used by `Parse`/`TryParse`:

```
var
  Lap1, Lap2, Diff: TTimeSpan;
  Msg: string;
begin
  Lap1 := TTimeSpan.FromMinutes(2.5);
  Lap2 := TTimeSpan.FromMinutes(2.25);
  Diff := Lap2 - Lap1;     //subtract one TTimeSpan from another
  if Lap2 > Lap1 then      //compare one TTimeSpan to another
    Msg := Format('Lap 2 was slower than lap 1 by %g secs',
      [Diff.TotalSeconds])
  else if Lap2 < Lap1 then //compare one TTimeSpan once more
    Msg := Format('Lap 2 was quicker than lap 1 by %g secs',
      [-Diff.TotalSeconds])
  else
    Msg := 'Lap 2 was the same time than lap 1';
  //add one TTimeSpan to another and convert result to a string
  Msg := Msg + SLineBreak +
    'Total time (hrs, mins, secs) was ' + (Lap1 + Lap2);
```

When using the implicit string conversion, you may need to be a bit careful with brackets. For example, the lack of brackets in the following means the two lap times are independently converted to strings, rather than the lap times added together as numbers first *then* converted to a string:

```
Msg := 'Total time (hrs, mins, secs) was ' + Lap1 + Lap2;
```

As with `Parse` and `TryParse`, localised settings aren't used when assigning to a string. If you want to use the user's localised time separator character rather than a colon, for example, then you'll just have to code it yourself:

```
function TimeSpanToLocalizedStr(const TS: TTimeSpan): string;
var
  I: Integer;
begin
  Result := TS;
  if FormatSettings.TimeSeparator = ':' then Exit;
  for I := 1 to Length(Result) do
    if Result[I] = ':' then Result[I] := FormatSettings.TimeSeparator;
end;
```

Locale-specific time separators are much rarer than locale-specific decimal or thousand separators however.

Timing things (TStopwatch)

The `TStopwatch` type, defined in the `System.Diagnostics` unit, is a simple record type for timing things. Its public interface is this:

```
class function Create: TStopwatch; static;
class function StartNew: TStopwatch; static;
procedure Reset;
procedure Start;
procedure Stop;
property Elapsed: TTimeSpan read GetElapsed;
property ElapsedMilliseconds: Int64 read GetElapsedMilliseconds;
property ElapsedTicks: Int64 read GetElapsedTicks;
property IsRunning: Boolean read FRunning;
class function GetTimeStamp: Int64; static; //get 'tick count'
class property Frequency: Int64 read FFrequency;
class property IsHighResolution: Boolean read FIsHighResolution;
```

To create a new TStopwatch instance, call either the Create or StartNew class method. StartNew does the same as Create, only setting the 'running' state to True; if you use Create instead, you will therefore have to call Start explicitly at some point. Whether 'running' or not, use one of the Elapsed properties to read off the time taken or taken up to now, and Stop to set the running state to False. Reset then sets the internal counter back to zero.

Here's an example of TStopwatch in action, being used to compare the speed of integer addition with integer multiplication:

```
uses
  System.SysUtils, System.TimeSpan, System.Diagnostics;

const
  LoopToNum = $FFFF;

function AddNums: Integer;
var
  I, J: Integer;
begin
  for I := 1 to LoopToNum do
    for J := 1 to LoopToNum do
      Result := I + J;
end;

function MultiplyNums: Integer;
var
  I, J: Integer;
begin
  for I := 1 to LoopToNum do
    for J := 1 to LoopToNum do
      Result := I * J;
end;

type
  TTestFunc = function : Integer;

procedure DoTiming(const Desc: string; const TestFunc: TTestFunc);
var
  Stopwatch: TStopwatch;
begin
  Stopwatch := TStopwatch.StartNew;
  TestFunc;
  Stopwatch.Stop;
  WriteLn(Format('%s took %n seconds',
    [Desc, Stopwatch.Elapsed.TotalSeconds]));
end;

begin
  DoTiming('Adding numbers', AddNums);
  DoTiming('Multiplying numbers', MultiplyNums);
end.
```

TStopwatch accuracy

On OS X, TStopwatch will always have a high resolution, meaning it will work to a very high level of accuracy. On Windows it may not however: in the first instance TStopwatch tries to use the Windows API's high resolution timer, and if that isn't available, falls back to the GetTickCount API function, which will definitely be around. Read the IsHighResolution static property to tell whether the preferred option is available and Frequency to tell what resolution (i.e., level of accuracy) is being used. The value reported will be the number of stopwatch ticks per second:

```
WriteLn('Does TStopwatch operate with a high resolution? ',
  TStopwatch.IsHighResolution);
WriteLn('TStopwatch has a resolution of ',
  TStopwatch.Frequency, ' ticks per second');
```

Converting between numerical units

The `System.ConvUtils` unit implements an extensible system for converting between measurements of different units. The core function is `Convert`:

```
function Convert(const AValue: Double; const AFrom,
  ATo: TConvType): Double;
function Convert(const AValue: Double; const AFrom1, AFrom2,
  ATo1, ATo2: TConvType): Double;
```

This is used like this:

```
uses
  System.SysUtils, System.ConvUtils, System.StdConvs;

var
  DistanceInMiles, DistanceInKM: Double;
begin
  DistanceInMiles := 30;
  DistanceInKM := Convert(DistanceInMiles, duMiles, duKilometers);
  WriteLn('A distance of ', FloatToStr(DistanceInMiles),
    ' miles is ', FloatToStr(DistanceInKM), ' kilometres');
end.
```

The second, more complicated variant of `Convert` is for cases when there is some sort of ratio:

```
uses
  System.SysUtils, System.ConvUtils, System.StdConvs;

var
  OldValue, NewValue: Double;
begin
  OldValue := 80.5;
  NewValue := Convert(OldValue, duNauticalMiles, tuHours,
   duYards, tuSeconds);
  Writeln(Format('%n nautical miles per hour translates to ' +
    '%n yards per second', [OldValue, NewValue]));
end.
```

`System.StdConvs` provides `TConvType` values for the following sorts of measurement unit ('conversion families'):

- Distance, prefixed with `du`, e.g. `duInches`, though also `duMicromicrons`, `duAngstroms`, `duMillimicrons`, `duMicrons`, `duMillimeters`, `duCentimeters`, `duDecimeters`, `duMeters`, `duDecameters`, `duHectometers`, `duKilometers`, `duMegameters`, `duGigameters`, `duFeet`, `duYards`, `duMiles`, `duNauticalMiles`, `duAstronomicalUnits`, `duLightYears`, `duParsecs`, `duCubits`, `duFathoms`, `duFurlongs`, `duHands`, `duPaces`, `duRods`, `duChains`, `duLinks`, `duPicas` and `duPoints`,
- Area, prefixed with `au`, e.g. `auHectares`, though also `auSquareMillimeters`, `auSquareCentimeters`, `auSquareDecimeters`, `auSquareMeters`, `auSquareDecameters`, `auSquareHectometers`, `auSquareKilometers`, `auSquareInches`, `auSquareFeet`, `auSquareYards`, `auSquareMiles`, `auAcres`, `auCentares`, `auAres` and `auSquareRods`.
- Volume, prefixed with `vu`, e.g. `vuCubicMillimeters`, though also `vuCubicCentimeters`, `vuCubicDecimeters`, `vuCubicMeters`, `vuCubicDecameters`, `vuCubicHectometers`, `vuCubicKilometers`, `vuCubicInches`, `vuCubicFeet`, `vuCubicYards`, `vuCubicMiles`, `vuMilliLiters`, `vuCentiLiters`, `vuDeciLiters`, `vuLiters`, `vuDecaLiters`, `vuHectoLiters`, `vuKiloLiters`, `vuAcreFeet`, `vuAcreInches`, `vuCords`, `vuCordFeet`, `vuDecisteres`, `vuSteres`, `vuDecasteres`, `vuFluidGallons`, `vuFluidQuarts`, `vuFluidPints`, `vuFluidCups`, `vuFluidGills`, `vuFluidOunces`, `vuFluidTablespoons`, `vuFluidTeaspoons`, `vuDryGallons`, `vuDryQuarts`, `vuDryPints`, `vuDryPecks`, `vuDryBuckets`, `vuDryBushels`, `vuUKGallons`, `vuUKPottles`, `vuUKQuarts`, `vuUKPints`, `vuUKGills`, `vuUKOunces`, `vuUKPecks`, `vuUKBuckets` and `vuUKBushels`. The `vuFluidXXX` and `vuDryXXX` values are for American amounts, `vuUKXXX` for the British Imperial measures.
- Mass, prefixed with `mu`, e.g. `muGrams`, but also `muMicrograms`, `muMilligrams`, `muCentigrams`, `muDecigrams`, `muDecagrams`, `muHectograms`, `muKilograms`, `muMetricTons`, `muDrams`, `muGrains`, `muLongTons`, `muTons`, `muOunces`, `muPounds` and `muStones`.
- Temperature, prefixed with `tu`, e.g. `tuCelsius`, but also `tuFahrenheit`, `tuKelvin`, `tuRankine` and `tuReaumur`.
- Time, also prefixed with with `tu`, e.g. `tuMilliSeconds`, but also `tuSeconds`, `tuMinutes`, `tuHours`, `tuDays`, `tuWeeks`, `tuFortnights`, `tuMonths`, `tuYears`, `tuDecades`, `tuCenturies`, `tuMillennia`, `tuDateTime` (for raw `TDateTime` values), `tuJulianDate` and `tuModifiedJulianDate`.

When calling `Convert`, the unit types must be of the same conversion family. As the compiler cannot check this statically, unmatched types will cause an `EConversionError` exception at runtime. For example, the following will output `EConversionError: Incompatible conversion types [Miles, Fahrenheit]`:

```
uses
  System.SysUtils, System.ConvUtils, System.StdConvs;

var
  NewValue: Double;
begin
  try
    NewValue := Convert(98, duMiles, tuFahrenheit);
    Writeln('Succe- what the...?');
  except
    on E: Exception do
      WriteLn(E.ClassName, ': ', E.Message);
  end;
end.
```

If you want to test whether two codes are compatible *before* calling `Convert`, call `CompatibleConversionTypes`:

```
uses
  System.SysUtils, System.ConvUtils, System.StdConvs;

begin
  WriteLn('Celsius to seconds: ',
    CompatibleConversionTypes(tuCelsius, tuSeconds));    //FALSE
  WriteLn('Celsius to Fahrenheit: ',
    CompatibleConversionTypes(tuCelsius, tuFahrenheit)); //TRUE
end.
```

Conversion families

Each conversion family is allocated its own identifying number, like each conversion type. For the standard families, these are assigned to the `cbDistance`, `cbArea`, `cbVolume`, `cbMass`, `cbTemperature` and `cbTime` variables of `System.StdConvs`. It is also possible to register both a new unit with a standard family, and an entirely new family using the `RegisterConversionType` and `RegisterConversionFamily` functions:

```
type
  TConversionProc = function(const AValue: Double): Double;

function RegisterConversionType(const AFamily: TConvFamily;
  const ADescription: string; const AFactor: Double): TConvType;
function RegisterConversionType(const AFamily: TConvFamily;
  const ADescription: string; const AToCommonProc,
  AFromCommonProc: TConversionProc): TConvType;

function RegisterConversionFamily(
  const ADescription: string): TConvFamily;
procedure UnregisterConversionFamily(const AFamily: TConvFamily);
```

Each family has a designated 'basic' unit type, though which conversions between other types go via. For example, if converting between miles per hour and kilometres per hour, the distances will be internally converted to metres first in order to make them directly comparable.

The basic types for the standard conversion families are as thus:

- The basic distance unit is metres.
- The basic area unit is square metres.
- The basic volume unit is cubic metres.
- The basic mass unit is grammes.
- The basic temperature unit is degrees Celsius.
- The basic time unit is days (like `TDateTime`).

Custom conversion types

In the simplest case, a custom unit would be registered against an existing family and have a relationship with the existing base type that is a simple ratio. To implement a custom unit with these attributes, create a new unit, add a global variable typed to `TConvType` in the interface section, call `RegisterConversionType` in the unit's initialization section, and `UnregisterConversionType` in its finalization section. In the following example, we do all this in order to add Roman miles as a distance type:

```
unit MyConvs;

```

```
interface

uses
  System.ConvUtils, System.StdConvs;

var
  duRomanMile: TConvType;

implementation

initialization
  duRomanMile := RegisterConversionType(cbDistance, 'Roman mile', 1479);
finalization
  UnregisterConversionType(duRomanMile);
end.
```

Our custom unit type will then be usable in the same way the standard ones are:

```
var
  Amount: Double;
begin
  Amount := Convert(2, duRomanMile, duMiles);
  WriteLn(Format('2 Roman miles = %g statute miles', [Amount]));
```

If the relation between custom and base type is not a simple ratio, then define a pair of global functions to perform the conversions. The following example does this to add ‘gas marks’ as a custom temperature unit. Gas marks were, and sometimes still are, used for cooking food in the United Kingdom and former countries of the British Empire; gas mark 1 is 275° Fahrenheit, with gas mark 2 300° F, gas mark 3 325° F and so on. However, gas mark 1/4 is 225° F and gas mark 1/2 is 250° F. Since the base temperature unit is degrees Celsius not degrees Fahrenheit, we also need to perform some multiplication and division:

```
var
  tuGasMark: TConvType;

function GasMarkToCelsius(const AValue: Double): Double;
function CelsiusToGasMark(const AValue: Double): Double;

implementation

function GasMarkToCelsius(const AValue: Double): Double;
begin
  if AValue >= 1 then
    Result := (243 + 25 * (AValue - 1)) * 5 / 9
  else
    Result := (168 + 100 * AValue) * 5 / 9;
end;

function CelsiusToGasMark(const AValue: Double): Double;
begin
  if AValue >= 135 then
    Result := ((AValue * 9 / 5 - 243) / 25) + 1
  else
    Result := ((AValue * 9 / 5) - 168) / 100;
end;

initialization
  tuGasMark := RegisterConversionType(cbTemperature, 'Gas mark',
    GasMarkToCelsius, CelsiusToGasMark);
finalization
  UnregisterConversionType(tuGasMark);
end.
```

As before, the new conversion type is used in the normal way once it has been registered:

```
var
  DegF, DegC: Double;
  I: Integer;
begin
  //convert gas mark 1/4
  DegF := Convert(0.25, tuGasMark, tuFahrenheit);
  DegC := Convert(0.25, tuGasMark, tuCelsius);
  WriteLn(Format('Gas mark 1/4 is %g°F or %.0f°C', [DegF, DegC]));
  //convert gas mark 1/2
  DegF := Convert(0.5, tuGasMark, tuFahrenheit);
  DegC := Convert(0.5, tuGasMark, tuCelsius);
  WriteLn(Format('Gas mark 1/2 is %g°F or %.0f°C', [DegF, DegC]));
  //convert gas marks 1-9
  for I := 1 to 9 do
  begin
    DegF := Convert(I, tuGasMark, tuFahrenheit);
    DegC := Convert(I, tuGasMark, tuCelsius);
    WriteLn(Format('Gas mark %d is %g°F or %.0f°C',
      [I, DegF, DegC]));
  end;
```

This outputs the following:

```
Gas mark 1/4 is 225°F or 107°C
Gas mark 1/2 is 250°F or 121°C
Gas mark 1 is 275°F or 135°C
Gas mark 2 is 300°F or 149°C
Gas mark 3 is 325°F or 163°C
Gas mark 4 is 350°F or 177°C
Gas mark 5 is 375°F or 191°C
Gas mark 6 is 400°F or 204°C
Gas mark 7 is 425°F or 218°C
Gas mark 8 is 450°F or 232°C
Gas mark 9 is 475°F or 246°C
```

System.ConvUtils helper functions

Beyond `Convert`, `System.ConvUtils` contains various helper routines. In each case, an exception is raised if the specified measurement types are not from the same family:

- `CompatibleConversionType` returns whether the specified type is in the specified family. Thus, `CompatibleConversionType(tuCelsius, cbTemperature)` returns `True` but `CompatibleConversionType(tuCelsius, cbDistance)` returns `False`.
- `ConvUnitInc` returns a value incremented by one or more units of the specified type. For example, `ConvUnitInc(24, tuHours, tuDays)` returns 48 (i.e., 24 hours plus one day), and `ConvUnitInc(24, tuHours, 0.5, tuDays)` returns 36 (i.e., 24 hours plus half a day).
- `ConvUnitDec` returns a value decremented by one or more units of the specified type.
- `ConvUnitAdd` adds two values of different measurement types together, returning in the units of a third. For example, `ConvUnitAdd(45, duCentimeters, 0.65, duMeters, duMillimeters)` returns 1,100 (45cm + 0.65m = 1,100mm).
- `ConvUnitDiff` returns the difference between two values of different measurement types, returning the result in units of a third.
- `ConvUnitSameValue` determines whether two values of different measurement types are equal. Since this boils down to a call to the `System.Math` unit's `SameValue` function, it allows for floating point rounding error.
- `ConvUnitCompareValue` compares two values of different types, returning 0 if the values are equal, -1 if the first is less than the second, and 1 if the first greater than the second. Thus, `ConvUnitCompareValue(5, duKilometers, 150, duMeters)` returns 1 since 5 kilometres is a longer distance than 150 metres.

3. OOP fundamentals

This chapter looks at the basics of Delphi's OOP implementation. Split into three parts, the first looks at encapsulation and the syntactical features shared between classes and records generally. This is followed by an overview of how inheritance and polymorphism work in Delphi. Finally, important Delphi specifics are looked at, such as the root class type (`TObject`), memory management and event properties.

Object rudiments in Delphi

In simple terms, an object is an entity in code that has data (its 'state') and accepts instructions. For example, when designing a user interface in Delphi, both forms and their controls (their buttons, labels, edit boxes, and so on) will be objects. Their state will include their captions or text, positions and sizes, and they will accept instructions to move, resize, change their text, receive the keyboard focus, and so on.

According to good OOP practice, an object's state should not be directly exposed to outside code in the way a global variable is directly exposed, since that would leave it open to outside interference. Instead, state should be 'encapsulated' (made private to the object), and publicly reachable (if publicly reachable at all) in a controlled fashion. In the case of a control, for example, exposing its caption directly would mean the text could be changed without the object knowing about it, leading to discrepancies between what the text should be and what is currently being displayed on the screen. Since the raw caption is encapsulated, however, changes to it are controlled so that the object can cause a repaint as appropriate.

Classes and records

In Delphi, the types whose instances can have encapsulated data and are able to receive instructions are 'classes' and 'records'. The data are declared as 'fields' and the instructions an object can receive are defined by its 'methods'. To instruct a record or class instance to do something, therefore, you 'call' the relevant method. Finally, encapsulation is achieved by declaring fields as 'private' using an appropriate visibility specifier, with a field's value exposed (if it is exposed at all) only indirectly via a public 'property'.

There is of course more to object oriented programming (OOP) than this, most notably the concepts of 'inheritance' and 'polymorphism'. In Delphi, these are only supported by classes, and indeed, when we talk of 'objects' in Delphi, we generally mean class instances specifically. Nonetheless, records can be used to define simple object types too — the `TFormatSettings`, `TTimeSpan` and `TStopwatch` types we met in the previous chapter illustrate this. Beyond a moderate level of complexity, record based objects won't be appropriate, however for basic things they allow for easier memory management: in a nutshell, where class instances must be explicitly 'freed', records do not, and where even the simplest class must be explicitly 'created' ('instantiated' if you're posh) before use, the simplest record does not.

Consequently, the present chapter will look first at OOP features shared by classes and records alike: specifically, the syntax for fields, methods and properties, the range of available visibility specifiers, and the possibility of declaring class and record 'helpers'. Class specific features will then be discussed following that.

Fields

A field is simply a variable made part of an object. Unlike when declaring regular variables, the `var` keyword is optional, and indeed isn't usually included in practice:

```
type
  TFoo = record
    StrField: string;
  end;

  TFooToo = class
    IntField: Integer;
    var AnotherField: Real;
  end;
```

In these cases, all instances of `TFoo` and `TFooToo` will have their own unique copy of the fields declared. For those situations where you wish for a field to be shared by all instances however, you can use the `class var` syntax, in which case the `var` keyword is compulsory:

```
type
  TFooAgain = record
    PersonalField: string;
    class var SharedField: string; //not 'record var'!
  end;

var
  Foo1, Foo2: TFooAgain;
begin
  Foo1.PersonalField := 'Foo 1''s personal field';
  Foo1.SharedField := 'Foo 1 has its say';
  Foo2.PersonalField := 'Foo 2''s personal field';
  Foo2.SharedField := 'Foo 2 has its say';
  WriteLn(Foo1.PersonalField);  //outputs Foo 1's personal field
  WriteLn(Foo1.SharedField);    //outputs Foo 2 has its say
  WriteLn(Foo2.PersonalField);  //outputs Foo 2's personal field
  WriteLn(Foo2.SharedField);    //outputs Foo 2 has its say
end.
```

Methods

A method is a procedure or function attached to an object, having access to that object's fields without qualification. The syntax is essentially identical to standalone routines, only with the method's declaration being done inside the declaration of the class or record type itself, and the implementation's header including the type name and a dot as a prefix to the method name:

```
type
  TFoo = record
    MyField: Integer;
    procedure MyProc;
  end;

  TFooToo = class
    function FuncWithParams(I: Integer; const S: string): string;
  end;

procedure TFoo.MyProc;
begin
  MyField := 42;
end;

function TFooToo.FuncWithParams(I: Integer; const S: string): string;
begin
  Result := IntToStr(I) + S;
end;
```

Inside each method lies an implicit `Self` variable. This provides a reference to the object instance itself, acting like the `this` keyword in C++, Java and C#, or the `Me` keyword in Visual Basic:

```
type
  TFoo = record
    S: string;
    procedure Test;
  end;

procedure TFoo.Test;
begin
  Self.S := 'When there are no name clashes, accessing';
  S := 'Self explicitly has the same effect as not doing so';
end;
```

Explicitly using `Self` like this is not idiomatic however since *needing* to do so indicates bad design: when you use an identifier, the compiler first tries to match it to something in the most immediate scope, then the second most immediate scope and so on. Inside a method, the items in the most immediate scope are the method's local variables and constants, followed by the fields and other methods of the object. Thus, if you *have* to use `Self`, it will be due to name clashes you have the power to avoid:

```
type
  TFoo = record
    WhatAmIDoing: string;
    procedure UpdateWhatAmIDoing;
  end;

procedure TFoo.UpdateWhatAmIDoing;
var
  WhatAmIDoing: string;
begin
  Self.WhatAmIDoing := 'Good question';
  WhatAmIDoing := 'Actually, this of course';
end;

var
  Foo: TFoo;
begin
  Foo.UpdateWhatAmIDoing;
  WriteLn(Foo.WhatAmIDoing); //outputs Good question
end.
```

Nonetheless, where `Self` plays a necessary role without indicating bad design is when a method needs to pass on a reference to its containing object:

```
type
  TFoo = record
    Value: string;
    procedure PassMeOn;
  end;

procedure WorkWithFooInstance(const Foo: TFoo);
begin
  WriteLn(Foo.Value);
end;

procedure TFoo.PassMeOn;
begin
  WorkWithFooInstance(Self);
end;
```

Static 'class' methods

Similar to how you can define fields that are shared by all instances of a class or record type, you can also define methods that are attached to the type itself rather than any particular instance. These can be called against either the type itself (`TMyType.MyStaticMethod`) or an instance of the type (`MyInst.MyStaticMethod`). They are declared by prefixing the `procedure` or `function` keyword with `class`, and appending the method declaration with the `static` directive:

```
type
  TMyUtils = record
    InstValue: Integer;
    class var Value: Integer;
    class procedure WriteValue; static;
  end;

class procedure TMyUtils.WriteValue;
begin
  WriteLn(Value);
end;

begin
  TMyUtils.Value := 42;
  TMyUtils.WriteValue;
end;
```

Notice the `static` keyword isn't added to the header of the method implementation. Also, while a static method has access to any 'class' fields, it won't have access to any instance ones (in the example, `InstValue`) since it isn't attached to any particular instance. Lastly, use of the `static` keyword is in fact optional in the case of classes. If missing, the method acquires an implicit `Self` variable, similar to ordinary 'instance' methods, but referencing the class type rather than the instance. We'll be looking at this feature under the rubric of 'metaclasses' in the next chapter.

Constructors

A further sort of method are 'constructors'. These are for initialising a new object's fields, and are declared using the constructor keyword in place of `procedure` or `function`. Usual practice is to name them `Create`, however in principle you can call them whatever you want:

```
type
  TIdiomaticObject = class
    constructor Create; overload;
    constructor Create(const AData: string); overload;
  end;

  TWeirdoRecord = record
    constructor MakeWithCharlatan(const AData: string);
  end;
```

Constructors are invoked by treating them as functions declared on the class or record type that return an instance of that type:

```
var
  Obj: TIdiomaticObject;
  Rec: TWeirdoRecord;
begin
  Obj := TIdiomaticObject.Create;
  Rec := TWeirdoRecord.MakeWithCharlatan('Tim Burgess');
```

In the case of a class, calling a constructor allocates the memory used by the instance, and is therefore a necessary step before using an object. To that effect, if you do not explicitly define any constructors yourself, your classes will inherit a default, parameterless constructor called `Create`.

In contrast, the memory for records is already allocated before you come to use them. Records therefore have no default constructor, since by default, there is nothing to 'construct'. Another difference with classes is that the compiler will not allow you to define a parameterless record constructor. However, it is possible to declare a static method that looks like one when used:

```
type
  TMyRec = record
    class function Create: TMyRec; static;
  end;

var
  Rec: TMyRec;
begin
  Rec := TMyRec.Create;
```

Unfortunately, it is not possible to force a record constructor to be called: in other words, the compiler allows code that uses a record type with one or more constructors defined to simply ignore them:

```
type
  TRecWithConstructor = record
    Data: string;
    constructor Create(const AData: string);
  end;

var
  Rec: TRecWithConstructor;
begin
  Rec.Data := 'This works without complaint!';
```

Properties

'Properties' are the way the internal state of an object is exposed to the outside world. In Delphi, they abstract away from both fields and methods, though appear to outside code as field-like things:

```
uses System.SysUtils; //for UpperCase (used below)

type
  TRecWithProps = record
  strict private
    FValue: string;
    procedure SetValue(const ANewValue: string);
  public
    property Value: string read FValue write SetValue;
  end;

procedure TRecWithProps.SetValue(const ANewValue: string);
begin
  FValue := UpperCase(ANewValue, loUserLocale); //force case
end;

var
  Rec: TRecWithProps;
begin
  Rec.Value := 'Hello';
  WriteLn(Rec.Value);                //HELLO
  Rec.Value := Rec.Value + ' world';
  WriteLn(Rec.Value);                //HELLO WORLD
end.
```

This example illustrates perhaps the most common case, in which reading the property is directly mapped to a backing field, and writing is mapped to a method that does some sort of validation or processing before assigning the field. Nonetheless, the read part can be delegated to a method too, and the write part configured to directly map to a field. Properties can also be read-only, in which case no `write` part appears in its declaration, or write-only, in which case no `read` part appears:

```
type
  TRecWithProps2 = record
  strict private
    FID: Integer;
    FText: string;
    FWeirdoProp: TDateTime;
    function GetValue: string;
  public
    property ID: Integer read FID;
    property Text: string read GetValue write FText;
    property WeirdoProp: TDateTime write FWeirdoProp;
  end;

var
  Rec: TRecWithProps2;
begin
  Rec.ID := 'Hello';       //won't compile!
  WriteLn(Rec.WeirdoProp); //won't compile!
```

While write-only properties involve perfectly valid syntax, they should generally be avoided — if a value can only be set, then it is much more idiomatic to expose a method directly, naming it either `SetXXX` or `UpdateXXX`.

Array properties

As normal properties look to the outside world like normal fields, so 'array properties' look like array fields. Unlike a normal property, an array property cannot be directly mapped to a field though, and so *must* have 'getter' and/or 'setter' methods:

```
type
  TRecWithArrayProp = record
  strict private
    FItems: array of string;
    function GetItem(AIndex: Integer): string;
    procedure SetItem(AIndex: Integer; const S: string);
  public
    property Items[Index: Integer]: string read GetItem write SetItem;
  end;

function TRecWithArrayProp.GetItem(AIndex: Integer): string;
begin
  Result := FItems[AIndex];
end;

procedure TRecWithArrayProp.SetItem(AIndex: Integer;
  const S: string);
begin
  FItems[AIndex] := S;
end;

var
  Rec: TRecWithArrayProp;
begin
  Rec.Items[0] := 'Nice';
```

If you so wish, an array property can have more than one 'dimension':

```
type
  TGraphicsSurface = class
  strict private
    function GetPixel(X, Y: Integer): UInt32;
    procedure SetPixel(X, Y: Integer; const ANewValue: UInt32);
  public
    property Pixels[X, Y: Integer]: UInt32 read GetPixel
      write SetPixel;
  end;
```

As a normal property only *looks* like a normal field, so an array property only looks like an array. Consequently, you cannot pass array properties to array standard functions like `Length`. If the data being exposed does have a finite length, normal practice is to expose a separate `Count` property:

```
property Count: Integer read GetCount;
```

On the other hand, not being a real array means the indexer type for an array property is unrestricted. For example, you might have an array property that takes a string for its indexer:

```
type
  TStringMap = class
  strict private
    function GetValue(const Name: string): string;
  public
    property Values[const Name: string]: string read GetValue;
  end;
```

Another feature of array properties is how for any given type, one array property can be designated the 'default' property. This means its name can be omitted when the property is read from or written to:

```
type
  TRecWithDefaultProp = record
  strict private
    function GetItem(AIndex: Integer): string;
    procedure SetItem(AIndex: Integer; const ANewValue: string);
  public
    property Items[Index: Integer]: string read GetItem
      write SetItem; default; //default directive added
  end;

var
  Rec: TRecWithDefaultProp;
begin
  Rec.Items[0] := 'Hello';    //normal version
  Rec[1] := 'World';          //short version
```

As shown here, the designated property has the `default` directive added to the end of its declaration. Be sure to include a semi-colon immediately before `default`, otherwise the compiler will complain about invalid syntax.

It is also possible to 'overload' the default property, which means the designated property appears multiple times with different parameters (the property name must be the same each time however):

```
type
  TRecWithDefaultProp2 = record
  strict private
    function GetItem(AIndex: Integer): string;
    procedure SetItem(AIndex: Integer; const ANewValue: string);
    function GetItemByName(const Name: string): string;
  public
    property Items[Index: Integer]: string read GetItem
      write SetItem; default;
    property Items[const Name: string]: string
      read GetItemByName; default;
  end;

var
  Rec: TRecWithDefaultProp2;
begin
  Rec[0] := 'Pretty';
  Rec['Neat'] := 'Stuff';
```

Visibility levels

Every member of a class or record type can be given an explicit visibility level. Both records and classes support `strict private`, `private` and `public` visibility; classes allow `strict protected`, `protected`, and `published` visibility too:

- `Strict private` means only methods of the class or record itself have access.
- This contrasts to mere `private` visibility, which also allows access to anything else in the same unit as well.
- `Strict protected` visibility means only methods of the class itself *or* any of its child classes have access.
- Similar to the `strict private`/`private` distinction, `protected` visibility, unlike `strict protected`, also allows access to anything else in the units of both the original class and its descendants.
- `Public` visibility any code has access.
- `Published` visibility is a special sort of `public` that has meaning for the component streaming system. Historically it meant runtime type information (RTTI) would be generated for the member concerned, but this is now generated for public members too by default. (RTTI itself will be looked at in chapter 12.)

While you can declare the visibilities of a type's members on an individual basis, the typical practice is to group them, starting with the lowest visibility first:

```
type
  TMyObject = class
  strict private
    FStrictPrivateField1: string;
    FStrictPrivateField2: Integer;
  private
    FPrivateField, FAnotherPrivateField: Byte;
  strict protected
```

```
    FStrictProtectedField: Char;
    procedure StrictProtectedMethod;
  protected
    procedure ProtectedMethod;
  public
    procedure PublicMethod;
  published
    property PublishedProp: Byte read FPrivateField write FPrivateField;
  end;
```

If a member is declared at the top of the type without any explicit visibility set, it is granted the widest, which will be either `public` or `published`.

As visibility blocks can appear in any order, so there can be more than one block for any one visibility level. In general, there is little point defining multiple blocks for the same level however.

Within a block, the compiler is a bit fussy over the order of fields and methods: specifically, once a method or property is defined, no more fields can be. As a result, the following will raise a compiler error:

```
type
  TMyObject = class
  protected
    procedure Method;
    Field: string;
  end;
```

To fix, either move `Field` to be above `Method`...

```
type
  TMyObject = class
  protected
    Field: string;
    procedure Method;
  end;
```

... or separate the declarations out into two blocks:

```
type
  TMyObject = class
  protected
    procedure Method;
  protected
    Field: string;
  end;
```

Since encapsulation is a Good Thing for maintainability, you should use as low a visibility as possible. In particular, the fields of a class should generally be declared with `strict private` visibility.

Nested types

Normally, types are declared at the unit level. However, it is possible to declare types *within* the declaration of a class or record type:

```
type
  TOuterType = record
    type
      TInnerType = (itFirst, itSecond);
    procedure Foo;
  end;
```

In this case, outside code will see the nested type as `TOuterType.TInnerType`:

```
var
  Enum: TOuterType.TInnerType;
begin
  Enum := itFirst;
```

If the nested type should be private to the outer type, then put it inside a `strict private` visibility block:

```
type
  TLinkedStringList = class
  strict private type
    PNode = ^TNode;
    TNode = record
      Value: string;
      Next: PNode;
    end;
  strict private
```

```
    FHeadNode, FTailNode: PNode;
  public
    procedure Add(const Value: string);
```

While there is no standard way of formatting nested type declarations (even the RTL, VCL and FMX sources don't agree!), I personally prefer separating them out into their own visibility blocks, as shown here.

In the case of a nested record or class type that involves methods, their implementation is coded as normal, only with an additional prefix of OuterTypeName. prepended to the method name:

```
type
  TOuterObject = class
  public type
    TInnerObject = class
      procedure Foo;
    end;
  end;

procedure TOuterObject.TInnerObject.Foo;
begin
  //...
end;
```

Class and record helpers

Any Delphi class or record type can have a 'helper' defined for it. This adds additional methods and properties that look as if they were part of the type's original definition from the point of view of calling code.

To give an idea of the syntax, consider the `TRect` type. This defines a simple record to hold positioning values (left, top, right and bottom), and is used in various places by the VCL. In XE2, helper methods for retrieving the implied width and height were added to the type, but prior to then, you could only retrieve these values by calculating them manually:

```
uses System.Types;

function GetRectWidth(const ARect: TRect): Integer;
begin
  Result := ARect.Right - ARect.Left;
end;

function GetRectHeight(const ARect: TRect): Integer;
begin
  Result := ARect.Bottom - ARect.Top;
end;

var
  R: TRect;
begin
  R.Left := 5; R.Top := 5; R.Right := 90; R.Bottom := 90;
  WriteLn('Rect size is ', GetRectWidth(R), ' x ', GetRectHeight(R));
```

Using a record helper however, `Width` and `Height` properties could have been added 'virtually':

```
uses Types;

type
  TRectHelper = record helper for TRect
  strict private
    function GetWidth: Integer;
    function GetHeight: Integer;
  public
    property Width: Integer read GetWidth;
    property Height: Integer read GetHeight;
  end;

function TRectHelper.GetWidth: Integer;
begin
  Result := Right - Left;
end;

function TRectHelper.GetHeight: Integer;
begin
  Result := Bottom - Top;
end;

var
  R: TRect;
begin
```

```
  R.Left := 5; R.Top := 5; R.Right := 90; R.Bottom := 90;
  WriteLn('Rect size is ', R.Width, ' x ', R.Height);
```

In practice, there are strict rules about what a helper can add to the helped type. In particular, no new fields can be defined, and no virtual method of a class can be overridden. Furthermore, only one helper for any given class or record type can be in scope. So, say both `UnitA` and `UnitB` define class helpers for `TStrings`, and `UnitC` uses both. Here, only the extra members added by the helper defined in `UnitB` will be available, assuming `UnitB` comes after `UnitA` in the uses clause — otherwise, only the helper members in `UnitA` will be 'seen'.

In the case of class helpers, this last limitation is slightly ameliorated by the fact one class helper can inherit from another:

```
type
  TObjectHelper = class helper for TObject
    procedure Foo;
  end;

  TObjectHelperEx = class helper(TObjectHelper) for TObject
    procedure Foo2;
  end;
```

This has the effect of adding to `TObjectHelperEx` all the members (here, the `Foo` method) of `TObjectHelper`. However, chaining helpers together like this doesn't make for a very elegant coding style.

In fact, one might go further and doubt the sense of using class and record helpers at all. Such a view might even be thought reflected in the online help, which says this about them:

```
Class and record helpers provide a way to extend a type, but they should not be viewed as a design tool to be used when
```

Nonetheless, online help still goes to the bother of properly documenting the feature, which seems odd if it isn't supposed to be used! So, how might helpers be used in a way that isn't just badly imitating proper OOP techniques?

One possible answer is to extend standard classes — especially core abstract base classes like `TStream` or `TStrings` — that you can't go ahead and rewrite yourself. For example, consider `TStream`, a core class we will be looking at in detail in chapter 8. As it currently stands, you can only read and write untyped buffers with it, and while reading and writing specific value types is easy enough, it does require ensuring you get the size parameter correct to `ReadBuffer` and `WriteBuffer` respectively. Given that, why not define a class helper for `TStream` that adds methods to read and write values of a specified value type? Here's a simple illustration of such a beast, which could be extended by more methods that read/write in 'big endian' format, for example:

```
uses
  System.Types, System.SysUtils, System.Classes;

type
  TStreamHelper = class helper for TStream
    function ReadValue<T: record>: T;
    procedure WriteValue<T: record>(const Value: T);
  end;

function TStreamHelper.ReadValue<T>: T;
begin
  ReadBuffer(Result, SizeOf(T));
end;

procedure TStreamHelper.WriteValue<T>(const Value: T);
begin
  WriteBuffer(Value, SizeOf(T));
end;
```

It can then be used like this:

```
procedure TestHelper;
var
  DT: TDateTime;
  R: TRect;
  Stream: TMemoryStream;
begin
  Stream := TMemoryStream.Create;
  try
    //write the current date/time and a TRect structure...
    Stream.WriteValue(Now);
    Stream.WriteValue(Rect(1, 2, 99, 100));
    //read them back
    Stream.Position := 0;
    DT := Stream.ReadValue<TDateTime>;
    R := Stream.ReadValue<TRect>;
    WriteLn('Date/time value written: ', DateTimeToStr(DT));
    WriteLn(Format('TRect structure written: (%d, %d)-(%d, %d)',
      [R.Left, R.Top, R.Right, R.Bottom]));
  finally
    Stream.Free;
  end;
end;
```

The alternative would be to define an equivalent set of global routines (or record type with an equivalent pair of 'class' methods) that take an additional `TStream` parameter to work on. Using a class helper, however, results in cleaner-looking code in my view. So long as inheritance or simple rewriting isn't a realistic option then, class and record helpers can play a useful — if limited — role.

Class specifics

So far in this chapter, we've been looking at classes and records pretty much interchangeably. In practice, classes are usually used far more than records, due to the fact records do not support key OOP features like inheritance and polymorphism.

Inheritance — defining the parent class

For one class to 'inherit' from another means that the second extends the first with additional and potentially amended behaviours; put crudely, a child class is a version of its parent that does (or has) more stuff. The syntax for declaring the parent type of a class is to specify the parent class' name inside brackets after the `class` keyword:

```
type
  TParent = class
    StrValue: string;
  end;

  TChild = class(TParent)
    IntValue: Integer;
  end;

procedure InitializeChild(Child: TChild);
begin
  Child.IntValue := 42;     //field introduced in TChild itself
  Child.StrValue := 'Test'; //field inherited from TParent
end;
```

If no explicit parent is given, then `TObject` — the ultimate ancestor of every Delphi class — becomes the parent. Like in Java or C#, classes in Delphi can only have one immediate parent; in the jargon, all these languages have a 'single inheritance' class model, in contrast to the 'multi inheritance' model had by C++.

The terms 'parent class', 'ancestor class', 'base class' and 'superclass' are all synonyms; likewise, so too are 'child class', 'descendant class' and 'subclass'. It doesn't really matter what terms you use, however the 'superclass' and 'subclass' formulations, which are prevalent in Java programming, are much less so in a Delphi context. This is probably because Delphi has traditionally been a Windows development tool, and 'subclassing' has a particular (and slightly different) meaning in Windows API programming. (As an aside, a 'subclass' is so-called because its instances form a subset of the instances of its 'superclass'. It is *not* a subclass in the sense it offers less functionality than the superclass — quite the contrary, the usual case is for functionality to increase the deeper into a class hierarchy you go.)

The 'is' and 'as' operators

The relationship between an ancestor class and its descendants is 'polymorphic' in the sense that instances of the child can be used wherever instances of the parent are expected:

```
procedure InitializeParent(Parent: TParent);
begin
  Parent.IntValue := 123;
end;

var
  Obj: TChild;
begin
  Obj := TChild.Create;
  try
    InitializeParent(Obj);
```

This is commonly expressed as the descendant and ancestor class forming an 'is a' relationship: `TChild` is a `TParent`. This notion works particularly well in the case of class hierarchies in GUI frameworks like the VCL or FireMonkey. For example, a text box (`TEdit`) is a control (`TControl`), which is a component (`TComponent`), which is a persistable object (`TPersistent`), which is an object (`TObject`).

At the language level, the `is` operator is available to test for such a relation. Talking an object reference as its first operand and a class reference as its second, this returns a `Boolean` indicating whether the class instance in question either has the specified type, or has a class descending from it. When the instance is `nil`, `is` returns `False`:

```
uses
  Vcl.Classes, Vcl.Controls, Vcl.StdCtrls;

var
  Obj: TObject;
begin
```

```
  Obj := nil;
  if Obj is TObject then
    WriteLn('Is an object')
  else
    WriteLn('Must be nil');
  Obj := TEdit.Create(nil);
  try
    WriteLn('Obj now assigned...');
    if Obj is TEdit then WriteLn('Is a text box');
    if Obj is TControl then WriteLn('Is a control');
    if Obj is TComponent then WriteLn('Is a component');
    if Obj is TPersistent then WriteLn('Is a persistable obj');
    if Obj is TObject then WriteLn('Is an object');
  finally
    Obj.Free;
  end;
  Obj := nil;
  if Obj is TObject then
    WriteLn('Is still an object')
  else
    WriteLn('Is not now an object');
end.
```

Once an `is` test returns `True`, it is safe to 'hard cast' to the destination type:

```
uses
  Vcl.SysUtils, Vcl.Classes, Vcl.Controls, Vcl.StdCtrls;

procedure UpdateText(Ctrl: TControl; const Str: string);
begin
  if Ctrl is TEdit then         //test for TEdit
    TEdit(Ctrl).Text := Str     //perform a hard cast to TEdit
  else if Ctrl is TMemo then    //test for TMemo
    TMemo(Ctrl).Lines.Add(Str)  //perform a hard cast to TMemo
  else
    raise EArgumentException.Create(
      'Expected either a TEdit or a TMemo');
end;
```

An alternative to the use of `is` followed by a hard cast is the `as` operator. Rather than returning a `Boolean`, this returns the input reference cast to the tested `type` if the test succeeds, or raises an exception if it doesn't. Unlike with `is`, the `as` operator treats a `nil` input as valid. This is likely to generate 'access violations' if you're not careful:

```
type
  TMyObject = class
    Data: string;
  end;

procedure Foo(Obj: TObject);
begin
  WriteLn((Obj as TMyObject).Data); //Obj being nil will raise
end;                                //an EAccessViolation

var
  MyObj: TMyObject;
begin
  try
    MyObj := TMyObject.Create;
    try
      MyObj.Data := 'Success';
      Foo(MyObj);
    finally
      MyObj.Free;
    end;
    MyObj := nil;
    Foo(MyObj);
  except
    on E: Exception do
      WriteLn(E.ClassName, ': ', E.Message);
  end;
end.
```

In simple terms, an access violation means you have attempted to reference a piece of memory that isn't valid for what you are trying to reference it for. While an access violation won't corrupt memory itself (in fact, avoiding such an eventuality is the reason access violation exceptions exist), you still shouldn't allow them to happen.

Virtual methods

By default, methods in Delphi are 'statically' bound. This impacts what happens when a child class introduces a method with a name and 'signature' (i.e., same set of parameters, and if a function, return type) that matches one introduced by an ancestor class, and an instance of the child is accessed polymorphically as an instance of the parent (which is to say, used by something expecting an instance of the parent):

```
type
  TParent = class
    procedure Foo(const Msg: string);
  end;

  TChild = class(TParent)
    procedure Foo(const Msg: string);
  end;

procedure TParent.Foo(const Msg: string);
begin
  WriteLn('TParent.Foo: ', Msg);
end;

procedure TChild.Foo(const Msg: string);
begin
  WriteLn('TChild.Foo: ', Msg);
end;

var
  Obj: TParent;
begin
  Obj: TParent;
begin
  Obj := TChild.Create;
  try
    Obj.Foo('test call');
  finally
    Obj.Free;
  end;
end.
```

Being statically bound, the method called in such a situation will be the one defined by the parent class, causing '`TParent.Foo`' to be outputted in the example.

The alternative is to use 'virtual' methods. In that case, a method declared in a base class may be 'overridden' by a descendant class:

```
type
  TParent = class
    procedure Foo(const Msg: string); virtual;
  end;

  TChild = class(TParent)
    procedure Foo(const Msg: string); override;
  end;
```

With the rest of the previous code remaining the same, '`TChild.Foo`' will now be outputted.

Calling inherited implementations

Typically, the override of a virtual method adds to what the original method does, rather than simply replacing the original implementation. When that pertains, use the `inherited` keyword to call the method that is being overridden:

```
procedure TChild.Foo(const Msg: string);
begin
  inherited Foo(Msg); //call the inherited implementation
  WriteLn('TChild.Foo: ', Msg);
end;
```

When the method being overridden is a procedure, as it is here, you can drop the method name and parameters, and just use the `inherited` keyword alone:

```
procedure TChild.Foo(const Msg: string);
begin
  inherited; //call inherited implementation with the same arg
  WriteLn('TChild.Foo: ', Msg);
end;
```

While the commonest case is for an override to call the inherited implementation first, that doesn't have to be the case. In fact, you can even call an inherited implementation outside of the method that overrides it:

```
type
  TChild = class(TParent)
    procedure Foo(const Msg: string); override;
    procedure AnotherProc;
  end;

procedure TChild.AnotherProc;
begin
  WriteLn('TChild.AnotherProc');
  inherited Foo('hello again');
end;
```

Abstract methods and classes

An 'abstract' method is a virtual method that has no implementation, requiring descendant classes to override it:

```
type
  TAncestor = class
    procedure Foo; virtual; abstract;
  end;
```

If you instantiate `TAncestor` and attempt to call `Foo`, an exception will be raised at runtime.

Along with individual methods, a whole class can be declared abstract. The compiler at present doesn't make anything of it though — if a warning is raised on instantiating an abstract class, it will be because an individual method is abstract, not the class as such:

```
type
  TMyObject = class abstract(TObject)
    procedure Foo;
  end;

procedure TMyObject.Foo;
begin
  WriteLn('TMyObject.Foo');
end;

var
  Obj: TMyObject;
begin
  Obj := TMyObject.Create; //causes no warning, let alone error
  try
    Obj.Foo;
  finally
    Obj.Free;
  end;
end.
```

The opposite of requiring a class to be descended from — i.e., disallowing a class to be descended from — can be done by using the `sealed` directive:

```
type
  TMyObject = class sealed(TObject)
  end;

  TMyChild = class(TMyObject) //compiler error
  end;
```

Attempt to descend from a class marked `sealed`, and a compiler error will result.

The circular uses clause problem

Imagine two units, prosaically named `Unit1` and `Unit2`. If `Unit2` appears in the interface section uses clause of `Unit1`, then Unit1 cannot appear in any interface section uses clause of `Unit2`:

```
unit Unit1;

interface

uses Unit2;

implementation
end.
```

```
unit Unit2;

interface

uses Unit1; //won't compile

implementation
end.
```

To fix this, either `Unit1`'s reference to `Unit2` must be moved to an implementation section uses clause or vice versa:

```
unit Unit2;

interface

implementation

uses Unit1; //now compiles

end.
```

Until you get used to it, this is pretty much guaranteed to annoy you at some point since it can prevent keeping mutually dependant classes in different code files.

Assuming you can't just move one or more unit references to the implementation section instead, the solution is to separate out the interface of one or both classes into either an abstract base class or an object interface. The latter can then be declared in a further unit that can be used by the main ones.

For example, imagine a class called `TParent` whose instances contain `TChild` objects. For whatever reason, the latter require knowledge of their parent. To keep things simple, let's just say they need to be able to call a method of the parent called `Foo`. To allow `TParent` and `TChild` to be in separate units, we can declare a third that defines the interface of `TParent` in the form of an abstract class:

```
unit MyClassIntf;

interface

type
  TParent = class
    procedure Foo; virtual; abstract;
  end;

implementation
end.
```

The unit `TChild` is defined in would then reference `MyClassIntf`:

```
unit ChildClass;

interface

uses
  MyClassIntf;

type
  TChild = class
  strict private
    FIndex: Integer;
    FParent: TParent;
  public
    constructor Create(AParent: TParent; AIndex: Integer);
    procedure DoSomething;
  end;

implementation

constructor TChild.Create(AParent: TParent; AIndex: Integer);
begin
  inherited Create;
  FIndex := AIndex;
  FParent := AParent;
end;

procedure TChild.DoSomething;
begin
  WriteLn('Child ', FIndex, ' calling Parent.Foo...');
  FParent.Foo;
  WriteLn('');
end;

end.
```

Since `ChildClass` doesn't use it, the unit for the parent class' concrete implementation can use `ChildClass` quite happily:

```
unit ParentClass;

interface

uses
  MyClassIntf, ChildClass;

type
  TConcreteParent = class(TParent)
  strict private
    FChildren: array[1..3] of TChild;
  public
    constructor Create;
    destructor Destroy; override;
    procedure Foo; override;
  end;

implementation

constructor TConcreteParent.Create;
var
  I: Integer;
begin
  inherited Create;
  for I := Low(FChildren) to High(FChildren) do
    FChildren[I] := TChild.Create(Self, I);
end;

destructor TConcreteParent.Destroy;
var
  I: Integer;
begin
  for I := Low(FChildren) to High(FChildren) do
    FChildren[I].Free;
  inherited Destroy;
end;

procedure TConcreteParent.Foo;
begin
  WriteLn('Parent says phooey to you too');
end;

end.
```

In this case, we could actually do away with the third unit entirely, and place the abstract `TParent` class in the `ChildClass` unit instead. However, if there to be another child class defined in another unit, we couldn't declare `TParent` twice, so the extra unit would be needed.

Core classes: TObject, TPersistent and TComponent

TObject

Every class in Delphi ultimately descends from `TObject`. This class introduces the following methods:

```
  { Do-nothing default constructor and destructor }
  constructor Create;
  destructor Destroy; virtual;
  procedure Free;
  { Extra construction and destruction hooks for descendant classes }
  procedure AfterConstruction; virtual;
  procedure BeforeDestruction; virtual;
  { Metaclass getter function }
  function ClassType: TClass;
  { Metaclass methods }
  class function ClassName: string;
  class function ClassNameIs(const Name: string): Boolean;
  class function ClassParent: TClass;
  class function InheritsFrom(AClass: TClass): Boolean;
  class function UnitName: string;
  { Key interface support method - used mainly internally }
  function GetInterface(const IID: TGUID; out Obj): Boolean;
  { General utility functions; default implementations aren't
    very interesting, but made virtual to allow descendants to
    return something useful }
  function Equals(Obj: TObject): Boolean; virtual;
  function GetHashCode: Integer; virtual;
  function ToString: string; virtual;
  { WinAPI-style message support methods, but can be used generally }
  procedure Dispatch(var Message); virtual;
  procedure DefaultHandler(var Message); virtual;
```

In `TObject` itself, all of `Create`, `Destroy`, `AfterConstruction` and `BeforeDestruction` don't actually do anything — they're just placeholders. Similarly, while `Equals`, `GetHashCode` and `ToString` *do* return something, they don't do anything particularly interesting: `Equals` just compares object references, `GetHashCode` just returns `Self` cast to an integer, and `ToString` just returns the class name. Nonetheless, by being declared virtual, they can be overridden to do (or provide) something more interesting, and a few standard classes do just that.

Of the class methods, `ClassName` and `ClassNameIs` do what they say on the tin — in the case of `TObject` itself, `ClassName` will therefore return `'TObject'`. Likewise, `UnitName` does what its name implies too, however in the case of a class declared in the project file (DPR) it will return the name of the program.

`InheritsFrom`, is used in the implementation of the `is` operator. Specifically,

```
if Obj is TMyObject then {do stuff} ;
```

is essentially a shortcut for

```
if (Obj <> nil) and Obj.InheritsFrom(TMyObject) then {do stuff} ;
```

`ClassParent` returns the parent type, giving a simple way of discovering a class' ancestry at runtime:

```
type
  TParent = class
  end;

  TChild = class(TParent)
  end;

  TGrandchild = class(TChild)
  end;

var
  Cls: TClass;
begin
  Cls := TGrandchild;
  repeat
    Write(Cls.ClassName, ' ');
    Cls := Cls.ClassParent;
  until (Cls = nil);
end. //outputs TGrandchild TChild TParent TObject
```

`GetInterface` outputs an interface reference for the specified interface, returning `False` if the object doesn't implement it. We will be looking at interfaces themselves in the next chapter, and in practice, you should generally use either the `as` operator or the `Supports` function from `System.SysUtils` rather than `GetInterface` directly.

Lastly, `Dispatch` and `DefaultHandler` provide some language support for Windows-style messaging independent of the Windows API itself. In use, each ‘message’ is assigned a unique, two byte number. The constants for numbers defined by the Windows API usually have a `WM_` prefix, those by the VCL a `CM_` one; in either case, the constants are conventionally written in all upper case:

```
const
  CM_SAYHELLO   = $0401;
  CM_SAYGOODBYE = $0402;
```

Aside from a number, each message also usually has a record type defined for it, though different messages can share a record type. The record must begin with a field typed to either `UInt16`/`Word` or `UInt32`/`LongWord`, which will be set at runtime to the message number. The Windows API uses `UInt32`/`LongWord` and assumes further fields that take the space of three pointer values, however the Delphi compiler doesn’t require that:

```
type
  TCMMyMessage = record
    MsgID: Word;
    SenderName: string;
  end;
```

The ordinary way to handle a message is to define a dedicated method for it. The method must use the `message` directive and take a single `var` parameter typed to the record. You can also override `DefaultHandler` to catch any message sent that hasn’t been assigned its own method to handle it (in a VCL control’s case, `DefaultHandler` is overridden to pass the message on to the `DefWindowProc` API function):

```
type
  TMessageObject = class
  strict protected
    procedure CMSayHello(var Msg: TCMMyMessage);
      message CM_SAYHELLO;
    procedure CMSayGoodbye(var Msg: TCMMyMessage);
      message CM_SAYGOODBYE;
  public
    procedure DefaultHandler(var Msg); override;
  end;

procedure TMessageObject.CMSayHello(var Msg: TCMMyMessage);
begin
  WriteLn('Hello ', Msg.SenderName);
end;

procedure TMessageObject.CMSayGoodbye(var Msg: TCMMyMessage);
begin
  WriteLn('Goodbye ', Msg.SenderName);
end;

procedure TMessageObject.DefaultHandler(var Msg);
begin
  WriteLn('Unrecognised message ID: ', UInt16(Msg));
end;
```

In use:

```
var
  Msg: TCMMyMessage;
  Obj: TMessageObject;
begin
  Msg.SenderName := 'frustrated WinAPI coder';
  Obj := TMessageObject.Create;
  try
    Msg.MsgID := CM_SAYHELLO;
    Obj.Dispatch(Msg);
    Msg.MsgID := CM_SAYGOODBYE;
    Obj.Dispatch(Msg);
    Msg.MsgID := CM_SAYGOODBYE + 1;
    Obj.Dispatch(Msg);
  finally
    Obj.Free;
  end;
end.
```

This program outputs ‘Hello frustrated WinAPI coder’, ‘Goodbye frustrated WinAPI coder’ and ‘Unrecognised message ID: 1027’.

In terms of its general functionality, object messaging in Delphi is a sort of precursor to object interfaces, in that it allows you to send the same instruction to objects with different ancestries. However, the big limitation of messaging is the lack of either compile time or runtime type checking: if you don't use the right record structure, then a hard-to-debug crash will be waiting in the wings to bite you. Interfaces, in contrast, are strongly typed, making them almost always the better option in practice.

Constructors and classes

As we saw earlier, a 'constructor' is a method that creates an object. Constructors for class-based objects are very flexible things: aside from being both overloadable and nameable to almost anything you fancy, they can also be virtual. More dubiously, it is also possible for a descendant class to completely ignore any constructor its parent introduced.

Nonetheless, the convention is still to stick to one name (`Create`) and call the inherited constructor in the very first line of your own:

```
constructor TMyObject.Create;
begin
  inherited Create;
  //do stuff...
end;
```

Since every class ultimately descends from `TObject`, `TObject` introduces a do-nothing constructor called `Create`, the snippet above will always be valid code: if neither the parent class nor any other descendant has a parameterless constructor called `Create`, `TObject`'s will get called. This works in exactly the same way as an `inherited` call would work in an ordinary method.

The syntax for declaring virtual and/or overloaded constructors follows that for ordinary methods too:

```
type
  TMyObject = class
    constructor CreateVirtual; virtual;
    constructor CreateVirtualAndAbstract; virtual; abstract;
    constructor CreateOverloaded; overload;
    constructor CreateOverloaded(AParam: Boolean); overload;
  end;
```

Since the rules for normal methods apply, virtual constructors of an ancestor class will by default be 'hidden' if a child class defines its own with the same name. The compiler will warn you of this when it happens; to remove the warning, do what you must do in the case of an ordinary method: either add the `reintroduce` directive to make clear you realise you're hiding an inherited member, or override the inherited constructor to be able to add the `override` directive to it.

For example, the `TComponent` class defines a virtual constructor called `Create` that takes an `Owner` parameter. If in a descendant class you wished to define an additional variant that took an extra parameter, you could either give it a different name...

```
type
  TMyComp = class(TComponent)
  public
    constructor CreateExt(AOwner: TComponent; Extra: Byte);
```

...or 'retrospectively' overload the inherited `Create`:

```
type
  TMyComp = class(TComponent)
  public
    constructor Create(AOwner: TComponent); overload; override;
    constructor Create(AOwner: TComponent; Extra: Byte); overload;

//...

constructor TMyComp.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
end;
```

Notice the override of the inherited constructor requires an implementation, though that implementation need only be to call the parent class' constructor of the same signature.

This again follows the pattern for ordinary methods, and there's a good reason for that: a Delphi constructor in itself is in fact not much more than an ordinary instance method. Consider the following code, which is formally valid if hardly advisable:

```
procedure TDodgy.Foo;
```

```
var
  I: Integer;
begin
  for I := 0 to 4 do
    Create;
end;
```

Contrary to what things may seem, this will not instantiate five new instances of `TDodgy`. Instead, it will just call `Create` five times on the *current* instance. To actually use a constructor *as* a constructor requires calling it against a class rather than an object reference. Typically, a class reference means a class identifier:

```
function TNotSoDodgy.Foo: TArray<TNotSoDodgy>;
var
  I: Integer;
begin
  SetLength(Result, 5);
  for I := 0 to 4 do
    Result[I] := TNotSoDodgy.Create;
end;
```

Unlike the previous snippet, this version *will* create five new instances of the class.

Destructors

As instances of a class must be explicitly constructed, so they must be explicitly destructed ('destroyed', 'freed'). Similar to constructors, destructors in Delphi can be named as you wish. However, unlike constructors, there's no sane reason to avoid the standard `Destroy`, which is introduced as a virtual method by `TObject`:

```
type
  TDestroyTest = class
  strict private
    FData: string;
  public
    destructor Destroy; override;
  end;

destructor TDestroyTest.Destroy;
begin
  //do custom finalisation here...
  inherited Destroy;
end;
```

As shown here, the usual pattern is to call the inherited implementation of `Destroy` at the *end* of the override, not its start.

Having created an object, you should not actually call `Destroy` to destroy it however. Rather, call `Free`, which checks the object reference isn't `nil` before calling `Destroy` itself. (We will come onto the importance of this shortly.) The basic usage pattern for locally-created objects is therefore this — the `try`/`finally` statement is used to ensure `Free` will be called even if an exception is raised:

```
procedure WorkWithObj;
var
  TempObj: TMyObject;
begin
  TempObj := TMyObject.Create;
  try
    //work with TempObj
  finally
    TempObj.Free;
  end;
end;
```

Note the object is created *outside* of the `try` statement. This is because the low-level RTL will ensure any exception raised during the constructor will cause `Destroy` to be called automatically, after which the exception is re-raised and nothing returned.

When more than one local object is created at the same time, you can set up multiple `try`/`finally` blocks like this:

```
procedure WorkWithTwoObjsA;
var
  Obj1, Obj2: TMyObject;
begin
  Obj1 := TMyObject.Create;
  try
    Obj2 := TMyObject.Create;
    try
      //work with the objects...
    finally
      Obj2.Free;
    end;
  finally
    Obj1.Free;
  end;
end;
```

Alternatively, you can define only the one, and initialise all but the first object reference to `nil` before the first `Create` call is made:

```
procedure WorkWithTwoObjsB;
var
  Obj1, Obj2: TMyObject;
begin
  Obj2 := nil;
  Obj1 := TMyObject.Create;
  try
    Obj2 := TMyObject.Create;
    //work with the objects...
  finally
    Obj2.Free;
    Obj1.Free;
  end;
end;
```

Setting the other references to `nil` prior to the `try` block is necessary because local variables aren't automatically initialised to anything, and `Free` can only tell the object doesn't exist if its reference is `nil`.

Object fields and Free

Using `Free` rather than `Destroy` directly is particularly important in the case of object fields that are themselves object references. Imagine, for example, a class called `TConvertor` that has a `TFileStream` field called `FOutputStream`. This sub-object is then created in `TConvertor`'s constructor, and destroyed in its destructor:

```
type
  TConvertor = class
  strict private
    FOutputStream: TFileStream;
  public
    constructor Create(const AOutputFileName: string);
    destructor Destroy; override;
    //other methods...
  end;

constructor TConvertor.Create(const AOutputFileName: string);
begin
  inherited Create;
  FOutputStream := TFileStream.Create(AOutputFileName, fmCreate);
end;

destructor TConvertor.Destroy;
begin
  FOutputStream.Destroy; //!!!should use FOutputStream.Free
  inherited Destroy;
end;
```

Say the `TFileStream` constructor raises an exception — for example, maybe the output file is currently open by another application that has blocked writes by anything else. In such a case,

- `TFileStream.Create` will abort and call `TFileStream.Destroy` automatically.
- `FOutputStream` won't be assigned, remaining `nil` (the fields of a class instance are automatically zero-initialised).
- Given the exception hasn't been caught by `TConvertor.Create`, the construction of the `TConvertor` object will itself be aborted, causing `TConvertor.Destroy` to be called automatically...
- ... at which point `FOutputStream.Destroy` will raise *another* exception, because `FOutputStream` remains `nil`.

Simply replacing the call to `FOutputStream.Destroy` with `FOutputStream.Free` will avoid this eventuality.

FreeAndNil

One small step beyond `Free` is `FreeAndNil`. Unlike `Free`, this isn't a method, but a freestanding procedure declared by the `System.SysUtils` unit. In use, you pass it an object reference. Internally, the procedure then assigns the reference to a temporary local variable, `nil`'s the parameter, and calls `Free` on the temporary (the indirection is to cover the case of a destructor that raises an exception). Because of some technicalities sounding parameters, the compiler will allow you to pass any sort of variable to `FreeAndNil`. Nonetheless, be careful to only ever pass an object reference:

```
uses
  System.SysUtils;

var
  Int: Integer;
  Obj: TStringBuilder;
begin
  Obj := TStringBuilder.Create;
  FreeAndNil(Obj); //OK
  FreeAndNil(Int); //this unfortunately compiles too!
end.
```

For such a simple routine, `FreeAndNil` has caused quite some controversy amongst experienced Delphi users. The gist of the case against is that actually *needing* to use it implies you're reusing the same object reference for different objects, and doing such a thing strongly suggests bad design. On the other hand, `nil`'ing old object references does, by definition, make it easy to check when objects have been destroyed — just test for `nil`. This can be useful for debugging purposes regardless of whether you intend to reuse object references or not. Quite simply, 'access violations' caused by nil object references are easy to identify, whereas the potentially random and possibly intermittent errors caused by a stale object reference are not.

Events

An 'event' in a Delphi context usually means an event property. This acts as a hook for assigning a method of another object, which then gets invoked when the event occurs. For example, the `TButton` class has an `OnClick` event, which allows arbitrary code to be run when the button is clicked.

What makes a property an event property is that it is typed to either a method pointer or a method reference type; conventionally, event properties also have an `On-` prefix, and are read/write. If you are writing a custom component to be installed into the IDE, they should also be given `published` visibility so that the user can assign them at design time via the 'Events' tab of the Object Inspector.

Method pointers are similar to regular procedural pointers, which we looked at in chapter 1, only with their instances being assignable to methods rather than standalone routines (hence their name), and their declarations having an `of object` suffix. For example, to declare an event type for events that take parameterless procedures for their handlers, you can use the following:

```
type
  TMyEvent = procedure of object;
```

The identifier used — here, `TMyEvent` — is arbitrary. Usually events have procedural method types; if there is a value to be returned, it is declared as a `var` or `out` parameter:

```
type
  TMyCallback = procedure (var Cancel: Boolean) of object;
```

Nonetheless, functions are possible too. For instance, here's a type for an event that takes a method with a `string` parameter and a `Boolean` result:

```
type
  TMyOtherEvent = function (const S: string): Boolean of object;
```

A common convention is to have a `Sender` parameter that denotes the object whose event it is. Since most events don't have any special parameters beyond that, this leads to the `TNotifyEvent` type (declared in `System.Classes`) being used a lot:

```
type
  TNotifyEvent = procedure (Sender: TObject) of object;
```

The point of making the `Sender` parameter loosely typed is so the same handler can be used for different types of object. For example, both a menu item and a button might share the same `OnClick` handler.

Once you have your event type, you can define the event property itself:

```
type
  TMyComp = class(TComponent)
  strict private
    FOnChange: TNotifyEvent;
  //...
  published
```

```
    property OnChange: TNotifyEvent read FOnChange write FOnChange;
  end;
```

In the case of objects only ever created at runtime, `public` visibility can be used; in the case of a custom component, `published` makes the event appear in the Object Inspector at design-time.

Once a class has declared an event property, it then needs to 'raise' the event at an appropriate point. This is done by checking whether the property (or its backing field) is assigned, before invoking the value as if it were a method itself:

```
if Assigned(FOnChange) then FOnChange(Self);
```

In the consuming code, 'handling' the event means assigning the event property with a method of the right signature (both the method and parameter names are immaterial). For a component being set up in the IDE, this can be done very easily via the Object Inspector. Nonetheless, it is hardly any more difficult in code:

```
type
  TMyForm = class(TForm)
  //...
  strict private
    FMyComp: TMyComp;
    procedure MyCompChanged(Sender: TObject);
  end;

procedure TMyForm.MyCompChanged(Sender: TObject);
begin
  ShowMessage('MyComp has changed!');
end;

procedure TMyForm.FormCreate(Sender: TObject);
begin
  FMyComp := TMyComp.Create(Self);
  //assign the OnChange event...
  FMyComp.OnChange := MyCompChanged;
end;
```

Frequently, events are assigned to ordinary 'instance methods' as above, in which the instance in question is either of a class or a record. However, they may also be assigned to 'class' methods, in which the 'instance' is a class *type*:

```
type
  TConsumer = class
    class procedure CompChange(Sender: TObject);
  end;

var
  Comp: TMyComp;
begin
  Comp := TMyComp.Create(nil);
  Comp.OnChange := TConsumer.CompChange;
```

This only works with true class methods though; their 'static' brethren (which is the only sort records support) cannot be used as event handlers:

```
type
  TConsumer2 = class
    class procedure CompChange(Sender: TObject); static;
  end;

  TConsumerRec = record
    class procedure CompChange(Sender: TObject); static;
  end;

var
  Comp: TMyComp;
begin
  Comp := TMyComp.Create(nil);
  Comp.OnChange := TConsumer2.CompChange;     //compiler error
  Comp.OnChange := TConsumerRec.CompChange;   //ditto
```

The reason is that static methods aren't really 'methods' at all internally, just standalone routines that are scoped to a certain type.

Method pointer internals

Internally, a method pointer is really a pair of pointers, one to the object or class type and one to the method as such. While you shouldn't really rely on the fact, this means any given method pointer can be typecast to the `TMethod` type, as declared by the `System` unit:

```
type
  TMethod = record
    Code, Data: Pointer;
  end;
```

Here, the `Code` field points to the procedure or function itself, and `Data` the object or class:

```
type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
  private
    procedure MyHandler(Sender: TObject);
  end;

//...

procedure TForm1.Button1Click(Sender: TObject);
var
  MethodRec: TMethod;
begin
  MethodRec.Code := @TForm1.MyHandler;
  MethodRec.Data := Self;
  Button2.OnClick := TNotifyEvent(MethodRec);
end;

procedure TForm1.MyHandler(Sender: TObject);
begin
  ShowMessage('You clicked the other button!');
end;
```

The sort of syntax demonstrated here isn't something that is generally needed. Indeed, `Button1Click` can (and should) be rewritten like this:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Button2.OnClick := MyHandler;
end;
```

Nonetheless, where `TMethod` can come into its own is in more advanced scenarios where the method address is looked up dynamically. For example, a framework might assign event handlers by polling the RTTI system for methods with certain names:

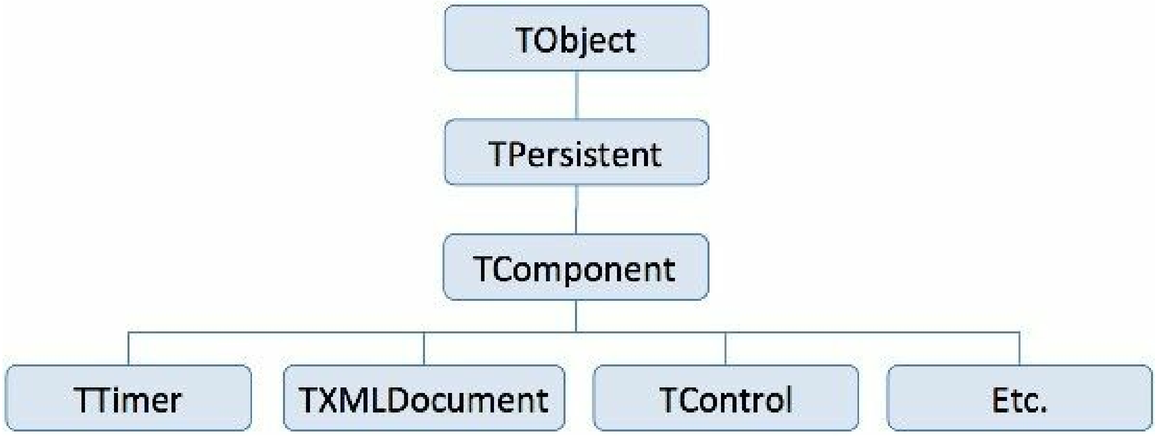
```
uses
  System.TypInfo, System.Rtti;

procedure THandlingObject.AssignClickHandlers(AMenu: TFmxObject);
var
  Context: TRttiContext;
  Method: TRttiMethod;
  Item: TFmxObject;
  Handler: TNotifyEvent;
begin
  for Method in Context.GetType(Self).GetMethods do
    if Method.MethodKind = mkProcedure then
    begin
      Item := AMenu.FindBinding(AMethod.Name);
      if (Item is TMenuItem) and
        not Assigned(TMenuItem(Item).OnClick) then
      begin
        TMethod(Handler).Code := AMethod.CodeAddress;
        TMethod(Handler).Data := Self;
        TMenuItem(Item).OnClick := Handler;
      end;
    end;
end;
```

In this FireMonkey example, menu items are assigned `OnClick` handlers on the basis of their `BindingName` property.

Components

In practical terms, a Delphi 'component' is an object with a class that can be added to the Tool Palette in the IDE. While many components are visual components ('controls') like buttons, edit boxes and so forth, they need not be. For example, `TTimer` and `TXMLDocument` are classes implemented as non-visual components.

Technically, a component is an object with a class ultimately descended from `TComponent`, as declared in `System.Classes`:



`TComponent` implements two features in particular. The first is the infrastructure for form streaming — the mechanism, in other words, that saves and restores a form's controls and property settings without any code having to be explicitly written. In the case of simple properties, it is enough to give them `published` visibility to have them participate in the streaming system:

```
uses
  System.SysUtils, System.Classes;

type
  TSimpleComponent = class(TComponent)
  strict private
    FText: string;
  published
    property Text: string read FText write FText;
  end;

function SaveStateToBytes(AComponent: TComponent): TBytes;
var
  Stream: TBytesStream;
begin
  Stream := TBytesStream.Create;
  try
    Stream.WriteComponent(AComponent);
    Result := Stream.Bytes;
  finally
    Stream.Free;
  end;
end;

procedure RestoreStateFromBytes(AComponent: TComponent;
  const AState: TBytes);
var
  Stream: TBytesStream;
begin
  Stream := TBytesStream.Create(AState);
  try
    Stream.ReadComponent(AComponent);
  finally
    Stream.Free;
  end;
end;

var
  Comp: TSimpleComponent;
  SavedState: TBytes;
begin
  Comp := TSimpleComponent.Create(nil);
  Comp.Text := 'This is a streaming test';
  SavedState := SaveStateToBytes(Comp);
  Comp.Text := 'What is this?';
  WriteLn(Comp.Text); //output: What is this?
  RestoreStateFromBytes(Comp, SavedState);
  WriteLn(Comp.Text); //output: This is a streaming test
  ReadLn;
end.
```

Component streaming will be looked at in much more detail in chapter 8.

Ownership pattern and free notifications

The second feature TComponent provides is an object 'ownership' pattern. Here, each TComponent instance may be assigned another to be its 'owner', so that when the owner is freed, the first object is automatically freed too. If an owned object is freed independently, things are still OK, since it will have notified its owner of the fact.

Due to the ownership pattern, any component you set up in the IDE does not need to be explicitly freed at runtime, since it will be owned (either directly or indirectly) by its form or data module. The same goes for any component you create at runtime yourself, so long as you have assigned an owner, and the owner itself either is freed explicitly at some point or has an owner that is freed and so on.

How the owner is set up in the first place is by passing a reference to the new object's constructor (you shouldn't attempt to assign the owner at a later point in the owned object's lifetime). In the following FireMonkey example, a form with a button is created at runtime without having been previously configured at designtime. Shown non-modally, the form is assigned the global Application object for its owner, and the button gets the form itself:

```
type
  TRuntimeForm = class(TForm)
  strict private
    FButton: TButton;
    procedure ButtonClick(Sender: TObject);
  public
    constructor Create(AOwner: TComponent); override;
  end;

procedure TRuntimeForm.ButtonClick(Sender: TObject);
begin
  Close;
end;

constructor TRuntimeForm.Create(AOwner: TComponent);
begin
  inherited CreateNew(AOwner);
  Position := TFormPosition.poScreenCenter;
  Width := 300;
  Height := 300;
  FButton := TButton.Create(Self); //Assign form as owner
  FButton.Parent := Self;          //Assign form as parent
  FButton.Position.X := (Width - FButton.Width) / 2;
  FButton.Position.Y := (ClientHeight - FButton.Height) / 2;
  FButton.Text := 'Click Me!';
  FButton.OnClick := ButtonClick;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  Form: TForm;
begin
  Form := TRuntimeForm.Create(Application);
  Form.Caption := 'Runtime Test';
  Form.Show;
end;
```

A VCL version of this code would be very similar.

It is also possible for one component to be notified when another is freed even if the first is not the second's owner. This is done by calling the second object's FreeNotification method, which causes the virtual Notification method of the first object to be called when the second comes to be freed. (If and when the first object no longer needs to know when the second has been destroyed, you can optionally call RemoveFreeNotification.) This feature is commonly used when implementing a property on one component that references a second. For example, the VCL TLabel control has a FocusControl property, which you assign a control like a TEdit to so that the label knows which control to focus when the user presses its accelerator key. If FocusControl were to be assigned and the edit box subsequently destroyed without the label learning of the fact, a stale reference and ultimately access violations would result.

The following demonstrates how to use FreeNotification:

```
uses
  System.SysUtils, System.Classes;

type
  TLinkedComponent = class(TComponent)
  strict private
    FBadRef, FGoodRef: TComponent;
    procedure SetGoodRef(Value: TComponent);
  protected
```

```
    procedure Notification(AComponent: TComponent;
      Operation: TOperation); override;
  public
    property BadRef: TComponent read FBadRef write FBadRef;
    property GoodRef: TComponent read FGoodRef write SetGoodRef;
  end;

procedure TLinkedComponent.Notification(AComponent: TComponent;
  Operation: TOperation);
begin
  inherited;
  if (AComponent = FGoodRef) then FGoodRef := nil;
end;

procedure TLinkedComponent.SetGoodRef(Value: TComponent);
begin
  if Value = FGoodRef then Exit;
  if FGoodRef <> nil then FGoodRef.RemoveFreeNotification(Self);
  if Value <> nil then Value.FreeNotification(Self);
  FGoodRef := Value;
end;

var
  Source: TComponent;
  Linked: TLinkedComponent;
begin
  Source := TComponent.Create(nil);
  Linked := TLinkedComponent.Create(nil);
  Linked.BadRef := Source;
  Linked.GoodRef := Source;
  Source.Free;
  WriteLn('GoodRef been nil''ed? ', Linked.GoodRef = nil); //TRUE
  WriteLn('BadRef been nil''ed? ', Linked.BadRef = nil);  //FALSE
  Linked.Free;
end.
```

In general, the pattern demonstrated by the `GoodRef` property here should be implemented whenever you declare a property on a component that references another component.

TPersistent

In between `TComponent` and `TObject` in the class hierarchy stands `TPersistent`, which like `TComponent` is declared in `System.Classes`. `TPersistent` is primarily designed for implementing object properties that participate in component streaming without being components themselves — examples include the `Font` and `Margins` properties of controls, which are typed to `TFont` and `TMargins` respectively. However, you can use `TPersistent` as a more general base class if you wish.

If you do, `TPersistent` has precisely one public method of interest, namely `Assign`:

```
procedure Assign(Source: TPersistent); virtual;
```

By default, if `Source` is `nil`, an exception is raised, otherwise `Source`'s protected `AssignTo` method is called. This however just raises an `EConvertError` exception, complaining that you cannot assign a `TMyClass` to a `TMyClass`! Consequently, without being overridden, `TPersistent.Assign` is not useful. However, its purpose is to set up a simple framework for making a 'deep copy' of an object (both `Assign` and `AssignTo` are virtual methods), and one in which the object being assigned may be ignorant of the class of the object being assigned from.

In the simplest case, both the source and destination type will be same class. In that situation, either `Assign` or `AssignTo` can be overridden. Choose `Assign`, and you can handle the case of assigning `nil` too, if that makes sense, and implement an ability to assign instances of other classes:

```
type
  TAssignable = class(TPersistent)
  strict private
    FData: string;
  public
    procedure Assign(Source: TPersistent); override;
    property Data: string read FData write FData;
  end;

procedure TAssignable.Assign(Source: TPersistent);
begin
  if Source = nil then
    FData := ''
  else if Source is TAssignable then
```

```
    FData := TAssignable(Source).Data
  else if Source is TStrings then   //declared in System.Classes
    FData := TStrings(Source).Text
  else
    inherited;
end;
```

When implementing `Assign`, you should always call `inherited` when the source class can't be matched, rather than raising an exception directly. This is to give the other class a chance to make the assignment itself. For example, as it stands, instances of our `TAssignable` class can be assigned string lists, but string lists cannot be assigned instances of our `TAssignable` class. To fix this, we can override `AssignTo` to provide support:

```
type
  TAssignable = class(TPersistent)
  //...
  protected
    procedure AssignTo(Dest: TPersistent); override;
  //...
  end;

procedure TAssignable.AssignTo(Dest: TPersistent);
begin
  if Dest is TStrings then
    TStrings(Dest).Text := FData
  else
    inherited;
end;
```

In use:

```
var
  Assignable: TAssignable;
  Strings: TStringList;
begin
  Assignable := TAssignable.Create;
  Strings := TStringList.Create;
  try
    Strings.Add('This is a test');
    Assignable.Assign(Strings);
    { The next statement will output 'This is a test' }
    WriteLn(Assignable.Data);
    Strings.CommaText := 'Some,other,values';
    { The next statement will output 'Some', 'other' and
      'values' on separate lines }
    WriteLn(Strings.Text);
    Strings.Assign(Assignable);
    WriteLn(Strings.Text);    //output: This is a test
  finally
    Assignable.Free;
    Strings.Free;
  end;
```

TRecall

When `Assign` is implemented, the `TRecall` helper class can be used to save and restore an object's state:

```
type
  TRecall = class
  //...
    constructor Create(AStorage, AReference: TPersistent);
    destructor Destroy; override;
    procedure Store;
    procedure Forget;
    property Reference: TPersistent read FReference;
  end;
```

To use, instantiate `TRecall` by passing a newly-created instance of the main class for `AStorage` and the instance whose state is being saved for `AReference`. Internally, the `TRecall` constructor will assign `AReference` to `AStorage`; when `TRecall` object is freed, it then assigns the other way so as to restore the original state:

```
var
  Assignable: TAssignable;
  SavedState: TRecall;
begin
  Assignable := TAssignable.Create;
  try
    Assignable.Data := 'This is the base state';
    SavedState := TRecall.Create(TAssignable.Create, Assignable);
    try
      Assignable.Data := 'Revised state';
      WriteLn(Assignable.Data); //output: Revised state
    finally
      SavedState.Free;
    end;
    WriteLn(Assignable.Data); //output: This is the base state
  finally
    Assignable.Free;
  end;
```

If something has happened that means the original state shouldn't be restored, call `Forget`; alternatively, if the original state should be forgotten about but only because the current state should be remembered instead, call `Store`:

```
var
  Assignable: TAssignable;
  SavedState: TRecall;
begin
  Assignable := TAssignable.Create;
  try
    Assignable.Data := 'This is the original state';
    SavedState := TRecall.Create(TAssignable.Create, Assignable);
    try
      Assignable.Data := 'Revised state';
      SavedState.Store;
      Assignable.Data := 'Further revision!';
      WriteLn(Assignable.Data); //output: Further revision!
    finally
      SavedState.Free;
    end;
    WriteLn(Assignable.Data);   //output: Revised state
  finally
    Assignable.Free;
  end;
```

4. Going further with classes and records

This chapter builds on its predecessor by considering Delphi's support for more advanced OOP features: metaclasses (sometimes called 'class references') and interfaces with regard to classes, operator overloading with regard to records, and generics (alias parameterised types) with regard to both. Also looked at will be anonymous methods (closures).

Metaclasses

When defining a class in Delphi, you at the same time implicitly define a 'metaclass', i.e., an instance of a class type. In creating a class hierarchy, a parallel metaclass hierarchy is therefore created too; and as all classes share an ultimate ancestor (namely `TObject`), so all metaclasses or 'class references' share an ultimate ancestor as well (namely `TClass`, the metaclass for `TObject`).

Compared to classes, metaclasses in Delphi are quite limited due to their being essentially stateless. This means you cannot really add fields to a metaclass. Nonetheless, class references can be assigned to variables and passed around, and fully support polymorphism in the sense virtual methods can be defined on one metaclass and overridden by the metaclasses of descendant classes.

Defining class methods

In terms of syntax, adding a method to a class' metaclass means declaring a method as normal, only adding a `class` prefix to it:

```
type
  TMyObject = class
    class procedure MyProc; virtual;
  end;

  TMyDescendant = class(TMyObject)
    class procedure MyProc; override;
  end;

class procedure TMyObject.MyProc;
begin
  WriteLn('The TMyObject metaclass says hi');
end;

class procedure TMyDescendant.MyProc;
begin
  WriteLn('The TMyDescendant metaclass says hi');
end;
```

To invoke a class method, you can call against an instance of the class or the class type directly:

```
procedure CallClassMethod;
var
  Inst: TMyObject;
begin
  Inst := TMyDescendant.Create;
  try
    Inst.MyProc;
  finally
    Inst.Free;
  end;
end;
```

Since `MyProc` was a virtual method overridden by `TMyDescendant`, the output here is `'The TMyDescendant metaclass says hi'` notwithstanding the fact `Inst` is typed to `TMyObject`.

Explicit class references

To retrieve the metaclass ('class reference') for a given object, you can call its `ClassType` function. This however always returns a `TClass` instance, i.e. the metaclass for `TObject`. To get a class reference to a specific `TClass` descendant, you must first declare a type for it using a syntax of

```
type
  MyMetaClassIdentifier = class of MyClass;
```

The identifier used can be anything you want, though the convention is to use a `TxxxClass` naming pattern:

```
type
  //TMyObject and TMyDescendant as before
  TMyObjectClass = class of TMyObject;

procedure CallClassMethodByParam(ClassType: TMyObjectClass);
begin
  ClassType.MyProc;
end;

procedure Test;
```

```
begin
  CallClassMethodByParam(TMyDescendant);
end;
```

For full effect, imagine `TMyObject`, `TMyObjectClass` and `CallClassMethodByParam` are in one unit and `TMyDescendant` and `Test` in another. In such a case, `CallClassMethodByParam` would have no knowledge of `TMyDescendant`. Despite this, it would end up calling the `MyProc` method of `TMyDescendant` rather than `TMyObject`, all thanks to the polymorphic nature of class references.

The implicit Self parameter

Similar to ordinary 'instance' methods, 'class' methods have an implicit `Self` parameter. Instead of pointing to an object instance however, this points to the class, which need *not* be the class in which the method is actually defined:

```
type
  TBase = class
    class procedure PrintClassName;
  end;

  TBaseClass = class of TBase;

  TDescendant = class(TBase)
  end;

class procedure TBase.PrintClassName;
begin
  WriteLn(ClassName); //or Self.ClassName
end;

var
  ClassRef: TBaseClass;
begin
  TDescendant.PrintClassName; //prints TDescendant *not* TBase
  ClassRef := TDescendant;
  ClassRef.PrintClassName;    //also prints TDescendant
  Readln;
end.
```

Notice `PrintClassName` did not have to be declared `virtual` for this behaviour to happen.

Static class members

Alongside class methods, Delphi also allows defining class constants, class variables, class properties, class constructors and class destructors. In the case of class variables and properties, the reuse of the word 'class' is perhaps a bit unfortunate, since they aren't actually part of the metaclass as such — instead, they're just globals with artificially restricted access. To complete the set (and to allow for 'class' properties with getters and setters), there are also 'static' class methods alongside class methods proper:

```
type
  TMyObject = class
  strict private
    class var FValue: Integer;
    class procedure SetValue(Value: Integer); static;
  public
    const MyClassConst = 42;
    class procedure AnotherStaticClassProc; static;
    class property Value: Integer read FValue write SetValue;
  end;
```

As a 'class' variable is really just a unit-level variable with restricted access, so a static 'class' method is really a global routine with restricted access. Unlike class methods proper, static class methods therefore have no implicit `Self` parameter:

```
procedure WorkWithMetaclass(ClassType: TClass);
begin
  //do something with ClassType...
end;

type
  TMyObject = class
    class procedure RealClassMethod;
    class procedure GlobalProcInDisguise; static;
  end;

class procedure TMyObject.RealClassMethod;
begin
```

```
  WorkWithMetaclass(Self); //compiles
end;

class procedure TMyObject.GlobalProcInDisguise;
begin
  WorkWithMetaclass(Self); //compiler error!
end;
```

Static members vs. metaclass members proper

In the case of a class with no descendants, the distinction between static members and members of the metaclass proper will be a distinction with very few practical consequences. Confusion can easily arise once inheritance hierarchies are introduced though. Consider the following program:

```
uses
  System.SysUtils;

type
  TBase = class
  strict private
    class var FID: Integer;
  public
    class property ID: Integer read FID write FID;
  end;

  TDescendantA = class(TBase)
  end;

  TDescendantB = class(TBase)
  end;

begin
  TDescendantA.ID := 22;
  TDescendantB.ID := 66;
  Writeln(TDescendantA.ID);
  Writeln(TDescendantB.ID);
  Readln;
end.
```

If you run it, you'll find 66 gets output twice! The reason is that `TDescendantA.ID` and `TDescendantB.ID` actually point to the same value, viz., `TBase.FID`. To have it otherwise requires a bit of plumbing, though it's not too difficult to do:

```
uses System.SysUtils, System.Generics.Collections;

type
  TBase = class
  strict private
    class var FIDs: TDictionary<TClass, Integer>;
    class constructor Create;
    class destructor Destroy;
  public
    class function ID: Integer;
    class procedure UpdateID(Value: Integer);
  end;

  TDescendantA = class(TBase)
  end;

  TDescendantB = class(TBase)
  end;

class constructor TBase.Create;
begin
  FIDs := TDictionary<TClass, Integer>.Create;
end;

class destructor TBase.Destroy;
begin
  FIDs.Free;
end;

class function TBase.ID: Integer;
begin
  if not FIDs.TryGetValue(Self, Result) then
  begin
    Result := 0;
    FIDs.Add(Self, Result);
  end;
end;

class procedure TBase.UpdateID(Value: Integer);
begin
  FIDs.AddOrSetValue(Self, Value);
end;

begin
  TDescendantA.UpdateID(22);
  TDescendantB.UpdateID(66);
  WriteLn(TDescendantA.ID);
  WriteLn(TDescendantB.ID);
  ReadLn;
end.
```

This then produces the output we originally expected, i.e. 22 followed by 66 (we will be looking at `TDictionary` itself, a class for mapping one thing to another, in chapter 6).

Class constructors and destructors

In Delphi, class constructors and class destructors are like a unit's `initialization` and `finalization` sections, only scoped to a metaclass (or record type, which can have them too). This means the code they contain will be automatically run at startup and closedown respectively. However, if the class type is never actually used, then the default behaviour is for the whole class to be 'smart linked' out of the final executable, in which case neither class constructor nor class destructor will be called.

As with normal constructors and destructors, class ones can be named as you like, with `Create` and `Destroy` being the conventional names. *Unlike* normal constructors and destructors though, class ones cannot be virtual, and moreover, cannot be called explicitly, regardless of what visibility you notionally give them (e.g. `protected` or `public` or whatever).

Not being virtual or explicitly callable gives rise to the question of what happens when both a base and descendant class defines a class constructor, since in the regular constructor case, this would imply only the constructor on the descendant class being called. With class constructors though, *both* will get executed, the constructor of the base class first. The converse will then happen with class destructors, as you might expect:

```
uses
  Vcl.Dialogs;

type
  TBase = class
    class constructor Create;
    class destructor Destroy;
  end;

  TDescendant = class(TBase)
    class constructor Create;
    class destructor Destroy;
  end;

class constructor TBase.Create;
begin
  ShowMessage('class constructor of TBase');
end;

class destructor TBase.Destroy;
begin
  ShowMessage('class destructor of TBase');
end;

class constructor TDescendant.Create;
begin
  ShowMessage('class constructor of TDescendant');
end;

class destructor TDescendant.Destroy;
begin
  ShowMessage('class destructor of TDescendant');
end;

begin
  TDescendant.ClassName; //ensure type isn't smart-linked out
end.
```

Running this program will produce four message boxes, from `TBase.Create`, `TDescendant.Create`, `TDescendant.Destroy` and `TBase.Destroy` respectively, and in that order.

Metaclass usage scenarios

While they are by no means as crucial a feature as classes themselves, metaclasses have their uses:

- They provide a simple way to annotate or describe classes, providing metadata that can be queried for at runtime. In this, the 'metadata' would take the form of one or more virtual class methods declared at the base of a class hierarchy (`Description`, `Capabilities`, etc.), which then get overridden by descendant classes.
- They can offer an elegant way to select between various classes without having to instantiate anything first. For example, in an image editing application, a series of classes might implement support for reading different file formats. In order to find out which class to use for a given file, an `IsFormatOK` class method could be defined and called.
- Virtual constructors and class references provide class 'factory' functionality that is built into the language.

To illustrate that last point, imagine a business application that provides a range of reports, with each report being implemented as its own `TForm` descendant. In the main form, the user is given the option of choosing which report or reports to generate. Tackled naïvely, this might be implemented using one big `if` statement:

```
if ReportToCreate = 'Salary report' then
  NewForm := TSalaryReport.Create(Self)
else if ReportToCreate = 'Pension report' then
  NewForm := TPensionReport.Create(Self)
else if ReportToCreate = ...
```

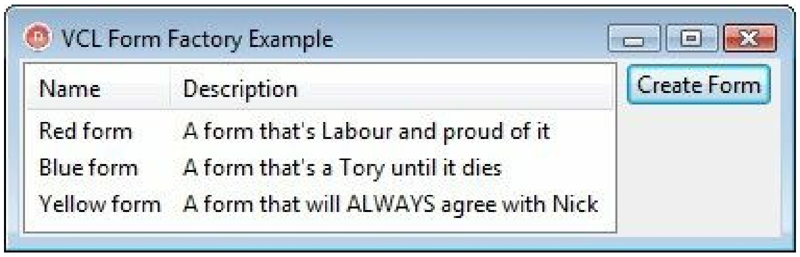
While this works OK with only a few report types to choose from, it will get increasingly messy and error prone once (say) half a dozen or more forms are available. Moreover, this approach requires the selection code to have knowledge of all the possible form classes 'up front', increasing unit interdependencies and necessitating code changes whenever the range of reports changes.

Using metaclasses in this situation allows for a more self-contained yet easily extendable approach. In a nutshell, the revised approach goes like this: firstly, a common abstract base class is defined, preferably with a virtual constructor (in the case of a form class, such a constructor will already be defined by virtue of the fact Delphi form classes ultimately descend from `TComponent`). Next, concrete descendants of this base class are defined and implemented, with the classes registering themselves with a central list when the application starts up. When the consuming code needs to instantiate a report class, it then polls the list of registered classes for an appropriate type. Class reference retrieved, an instance is then created.

In principle, nothing of the particular class being instantiated need be known — the fact it is a descendant of the base class should be enough. That's the theory anyhow; let's now look at the practice.

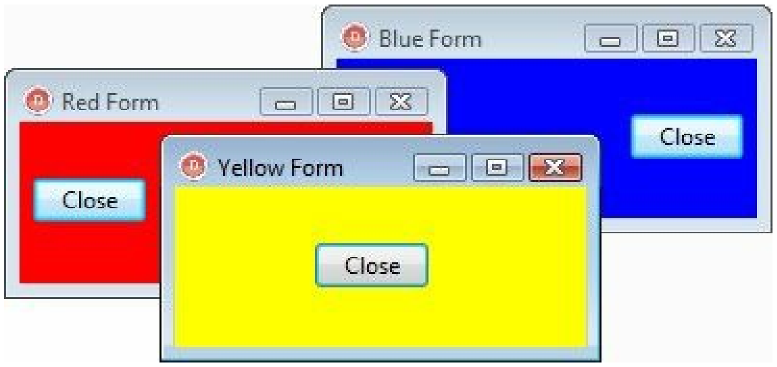
Class factories: an example

In this example, we'll be creating a simple application in which the main form displays a list of possible secondary forms to create:



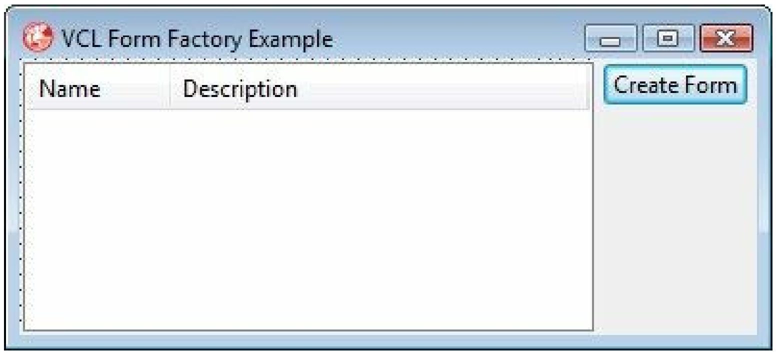
Name	Description
Red form	A form that's Labour and proud of it
Blue form	A form that's a Tory until it dies
Yellow form	A form that will ALWAYS agree with Nick

Keeping things simple, these secondary windows will just be forms with different colours rather than forms with functional differences:



Nonetheless, their classes will be added to a central registry so that the main form will be completely ignorant of where they are actually defined: on startup, it will just poll for what classes are available, and thereafter not distinguish between those that are returned.

To begin, create a new VCL forms application, add a `TListView` and `TPanel` to the main form, followed by a `TButton` to the panel. Call the controls `lsvForms`, `panRight` and `btnCreateForm` respectively, before arranging them in the designer to look like the following:



Here, the panel has its `Align` property set to `alRight`, with the list view's to `alClient`; the list view's `AlignWithMargins` property has also been set to `True`, and its `ViewStyle` set to `vsReport`. The columns have then been defined using the columns editor, invoked by double-clicking on the list view. Finally, the button's `Default` property has been set to `True`.

Next, we'll define the base class and registered form type list. To do this, add a new unit to the project (an easy way is to right click on the EXE's node in the Project Manager, select `Add New`, then `Unit`), and save it as `ClientForms.pas`. The base class will be a direct descendant of `TForm`, with the list of registered descendants being held as static data internal to the base class itself. This list will then be available for read-only access by external units.

To this effect, amend the new unit's interface section — i.e., its top half — to look like this:

```
unit ClientForms;

interface

uses
  System.Generics.Collections, Vcl.Controls, Vcl.Forms;

type
  TClientForm = class;

  TClientFormClass = class of TClientForm;

  TClientForm = class(TForm)
  private class var
    FRegisteredForms: TList<TClientFormClass>;
    class constructor Create;
    class destructor Destroy;
  protected
    class procedure RegisterForm;
  public
    class function Title: string; virtual; abstract;
    class function Description: string; virtual; abstract;
  end;

function RegisteredClientForms: TEnumerable<TClientFormClass>;
```

A few things may need explaining here. Firstly, notice how the explicit metaclass type is declared *before* the class itself, a fact that requires a 'forward declaration' of the class at the top (i.e., the `TClientForm = class;` line). This is because the metaclass is referred to within the class definition itself (viz., in the type of `FRegisteredForms`), and the Delphi compiler (following Pascal tradition) works in a 'top down' fashion.

Secondly, the return type of the `RegisteredClientForms` function may look a bit odd. The explanation is that `TList<T>` descends from `TEnumerable<T>`, and `TEnumerable<T>` defines only read-only access to the contained items. Since we want client units to only have read-only access to our registered class list, we can therefore simply expose it typed to its parent class to get the desired outcome.

Lastly, notice that a virtual constructor isn't defined, contrary to my earlier advice. The reason is that an ancestor class (specifically, `TComponent`) has already declared one. Consequently, there's no need to declare our own, notwithstanding the fact we could.

The implementation section of the unit (i.e., its bottom half) is trivial:

```
implementation

function RegisteredClientForms: TEnumerable<TClientFormClass>;
begin
  Result := TClientForm.FRegisteredForms;
end;

class constructor TClientForm.Create;
begin
  FRegisteredForms := TList<TClientFormClass>.Create;
end;

class destructor TClientForm.Destroy;
begin
  FRegisteredForms.Free;
end;

class procedure TClientForm.RegisterForm;
begin
  if not FRegisteredForms.Contains(Self) then
    FRegisteredForms.Add(Self);
end;

end.
```

On start-up, the main form can therefore call the `RegisteredClientForms` function, populating the list view with the information returned. In this, it will enumerate registered classes, retrieving the metadata added in the form of virtual class functions — the client forms' titles and descriptions. The reference to each metaclass itself can then be stored in the respective list view item's `Data` property, to be retrieved and used when the user clicks the Create Form button.

Let's now add some code behind the main form. Firstly, create an `OnCreate` event handler, either via the Events tab of Object Inspector or by double clicking on a blank area of the form in the designer. Implement the handler as thus:

```
procedure TfrmMain.FormCreate(Sender: TObject);
var
  FormClass: TClientFormClass;
  NewItem: TListItem;
begin
  lsvForms.Items.BeginUpdate;
  try
    for FormClass in RegisteredClientForms do
    begin
      NewItem := lsvForms.Items.Add;
      NewItem.Caption := FormClass.Title;
      NewItem.SubItems.Add(FormClass.Description);
      NewItem.Data := FormClass;
    end;
  finally
    lsvForms.Items.EndUpdate;
  end;
end;
```

To compile, you'll also need to add `ClientForms` to a uses clause — either do it manually or via `File|Use Unit`... (it can just go into the implementation section's uses clause).

Next we will define the button's `OnClick` handler. Create the stub by double clicking on the button, then amend the generated code to look like the following:

```
procedure TfrmMain.btnCreateFormClick(Sender: TObject);
var
```

```
  NewForm: TForm;
begin
  if lsvForms.SelCount = 0 then
  begin
    Beep;
    Exit;
  end;
  NewForm := TClientFormClass(lsvForms.Selected.Data).Create(Self);
  NewForm.Show;
end;
```

If you so wish, you can also assign this method (`btnCreateFormClick`) as the list view's `OnDblClick` handle. Either way, compile and run the application. All should be well and good, but for a lack of registered form classes!

Let's then define a few. Add a new form to the project as normal (e.g. by right-clicking on the EXE's node in the Project Manager, selecting `Add New`, then `Form`), before adding `ClientForms` to its interface section `uses` clause and manually editing its base class to `TClientForm`. The `Title` and `Description` class methods can (and should) then be overridden. Since the forms are being created on request, you should also ensure they don't appear in the application's auto-create list (`Project|Options`, `Forms`).

For the demo, just add a button called `btnClose` on each client form to close it:

```
unit RedClientForm;

interface

uses
  System.SysUtils, System.Classes, Vcl.Graphics, Vcl.Controls,
  Vcl.Forms, Vcl.StdCtrls, ClientForms;

type
  TfrmRedClient = class(TClientForm)
    btnClose: TButton;
    procedure btnCloseClick(Sender: TObject);
  public
    class function Title: string; override;
    class function Description: string; override;
  end;

implementation

{$R *.dfm}

class function TfrmRedClient.Title: string;
begin
  Result := 'Red form';
end;

class function TfrmRedClient.Description: string;
begin
  Result := 'A form that''s Labour and proud of it';
end;

procedure TfrmRedClient.btnCloseClick(Sender: TObject);
begin
  Close;
end;

initialization
  TfrmRedClient.RegisterForm;
end.
```

Notice the form is registered not in a class constructor but the unit's `initialization` section. The reason is simple: given the class is not explicitly referenced elsewhere, the class constructor would never be called!

You can find the full code for this demo in the book's source code repository (`http://delphi-foundations.googlecode.com/svn/trunk/`).

Interfaces

Object interface types (just 'interfaces' for short) allow classes to provide distinct 'views' of themselves, in which each view is composed of a certain set of methods and possibly properties too. Using interfaces, the same class is able to provide a variety of views for different purposes ('implement multiple interfaces'). Furthermore, different classes can offer the same view ('implement the same interface') even if the only ancestor class they share in common is `TObject`.

To switch metaphors, when a class implements an interface, it effectively signs a contract saying it will offer the service defined by the interface type. Significantly, the interface itself does not lend the class the means to do those things — it is just specifies the service. This contrasts to inheriting from another class, which involves the inheritance of both interface (the contract) and implementation (the means to perform it).

At first, this difference may suggest interface types half-baked. However, it is the key to their power: since interfaces completely decouple saying something can be done from specifying *how* it will be done, they can encourage cleaner, more flexible, and more testable code. Cleaner, because consumers of the classes involved are now forced to work with only the methods they should be needing; more flexible, because different classes can expose the same service without having to share a common base class; and more testable, because they can ease writing things like 'mocks', i.e. dummy implementations used for testing purposes.

These benefits (or potential benefits) go for interfaces in pretty much any language that has them that you may come to use. In Delphi, they also have a second rationale: since Delphi interfaces are reference counted where ordinary objects are not, they can ease memory management. However, this feature is not without its pitfalls, as we will learn later.

Defining an interface

As defined, an interface type looks a bit like a class or record declaration, only with the `interface` keyword used and anything related to implementation (e.g. fields and visibility specifiers) expunged. This leaves interface types being composed solely of methods and possible properties backed by methods:

```
type
  IOK = interface
    function GetMyReadWriteProp: Integer;
    procedure SetMyReadWriteProp(Value: Integer);
    function GetMyReadOnlyProp: string;

    procedure MyMethod;
    property MyReadOnlyProp: string read GetMyReadOnlyProp;
    property MyReadWriteProp: Integer read GetMyReadWriteProp
      write SetMyReadWriteProp;
  end;

  INotOK = interface
  strict private                          //won't compile!
    function GetMyReadOnlyProp: string;
  protected                               //won't compile!
    FMyField: Integer;                    //won't compile!
  public                                  //won't compile!
    class procedure Foo;                  //won't compile!
    property MyReadOnlyProp: string read GetMyReadOnlyProp;
  end;
```

Nonetheless, one interface can 'inherit' from another in the sense the methods of the base interface are automatically prepended to the methods of the descendant:

```
type
  IBase = interface
    procedure Hello;
  end;

  IChild = interface(IBase)
    procedure Goodbye;
  end;

procedure SayHello(const Intf: IChild);
begin
  Intf.Hello; //compiles
end;
```

If no parent interface is explicitly specified, the built-in `IInterface` type serves as the implied parent, similar to how `TObject` is the default parent type for classes.

GUIDs

While not strictly necessary, a quirk of interfaces in Delphi is that you should almost always provide them with a 'GUID' (globally unique identifier). This is done by placing a specially-formatted string of hexadecimal characters at the top of the interface type's definition. In the IDE, press `Ctrl+Shift+G` to generate a new GUID in the required format:

```
type
  IMyInt = interface
    ['{23D97BE8-E092-4F06-BE9C-481C4029A401}']
    procedure Hello;
  end;
```

The reason a GUID is usually needed is because it gives the interface a distinct identity. This allows you to query for it:

```
type
  INoGUID = interface
    procedure Foo1;
  end;

  IHasGUID = interface
  ['{54470F02-6531-4229-9CE5-B74EF6EC02E6}']
    procedure Foo2;
  end;

procedure Test(Intf: IInterface);
begin
  (Intf as IHasGUID).Foo2; //OK
  (Intf as INoGUID).Foo1;  //compiler error
end;
```

You should be careful not to assign the same GUID to more than one interface. The compiler won't complain, and will treat the two interfaces as interchangeable regardless of their actual methods!

Implementing an interface

Interfaces in Delphi can only be implemented by classes. In a class' header, the names of implemented interfaces follow the name of the parent class (if any), going in between the brackets that follow the `class` keyword:

```
type
  TMyObject = class(TParent, IIntf1, IIntf, IIntf3)
    //...
  end;
```

Since all interfaces ultimately descend from `IInterface`, and `IInterface` already defines three methods to implement (`_AddRef`, `_Release` and `QueryInterface`), the easiest way to implement an interface type is to have your class descend from `TInterfacedObject`, either directly or indirectly. `TInterfacedObject` is defined in the `System` unit, and implements the `IInterface` methods for you in a standard fashion.

Once a class has declared it implements an interface, it then needs to actually do so. This is done by ensuring all the methods that constitute the interface are found in either the class itself or one of its ancestors.

When referenced via an interface reference, these methods will be effectively public. However, inside the class itself, their visibility is irrelevant. In the following code, both `TFoo` and `TOtherFoo` therefore implement the `IFoo` interface in a legitimate manner:

```
type
  IFoo = interface
    ['{3419B526-C616-4DD0-8606-3166C92FBDBD}']
    procedure CallMe;
    procedure CallMeToo;
  end;

  TFoo = class(TInterfacedObject, IFoo)
  public
    procedure CallMe;
    procedure CallMeToo;
  end;

  TOtherFoo = class(TInterfacedObject, IFoo)
  strict private
    procedure CallMe;
    procedure CallMeToo;
  end;

procedure TFoo.CallMe;
```

```
begin
  WriteLn('TFoo says hi');
end;

procedure TFoo.CallMeToo;
begin
  WriteLn('TFoo says hi again');
end;

procedure TOtherFoo.CallMe;
begin
  WriteLn('TOtherFoo says hi');
end;

procedure TOtherFoo.CallMeToo;
begin
  WriteLn('TOtherFoo says hi again');
end;
```

In use:

```
var
  Obj: TOtherFoo;
  Intf: IFoo;
begin
  Obj := TOtherFoo.Create;
  try
    Intf := Obj as IFoo;
    Intf.CallMe; //OK
    Obj.CallMe;  //compiler error
    Intf := nil;
  finally
    Obj.Free;
  end;
end.
```

The fact `CallMe` can be invoked through the interface type isn't a bug, just a reflection of the fact an interface provides a distinct view of the objects that implement it.

Method resolution clauses

Normally, a class implements an interface by defining (or inheriting) methods with names and parameter lists that match those in the interface definition. However, you can also choose to explicitly hook up individual methods with 'method resolution clauses':

```
type
  TAnotherFoo = class(TInterfacedObject, IFoo)
    procedure CallYou;
    procedure CallYouToo;
    procedure IFoo.CallMe = CallYou;
    procedure IFoo.CallMeToo = CallYouToo;
  end;

procedure TFoo.CallYou;
begin
  //do something...
end;

procedure TFoo.CallYouToo;
begin
  //do something...
end;
```

In general, overuse of this feature can make code confusing to read. It can be a useful tool on occasion though, e.g. to avoid name clashes with inherited methods.

Inheriting interface implementations

If one class inherits from another, and the base class implements one or more interfaces, then the descendant class automatically implements those interfaces too.

In such a situation, the descendant class might wish to customise the inherited implementation. One way to do this would be for the parent class to make the relevant methods virtual, and so overridable in the normal way. However, another possibility is for the descendant class to simply make its support of the interface explicit. When that happens, any method needed by the interface found in the child class will override the corresponding method defined in the parent class *for* *the purposes of the interface*:

```
type
  //IFoo and TOtherFoo as before...

  TAnotherFoo = class(TOtherFoo, IFoo) { support IFoo explicitly }
  strict protected
    procedure CallMe;
  end;

procedure TAnotherFoo.CallMe;
begin
  WriteLn('Changed ya!');
end;
```

Note that `TAnotherFoo` doesn't have to redeclare `CallMeToo` as well, since the compiler can (and will) pick up the base class' implementation.

Reference counting

As a general rule, objects accessed through interface references should *only* be accessed through interface references. This is due to the fact interfaces are reference counted.

In a nutshell, the reference-counting scheme goes like this: when one interface reference is assigned to another, the internal usage counter of the object being assigned is incremented; and when an interface reference either goes out of scope or is assigned `nil`, this reference count is decremented. Once it falls to zero, the object is automatically freed:

```
type
  TTest = class(TInterfacedObject)
  public
    destructor Destroy; override;
  end;

destructor TTest.Destroy;
begin
  inherited;
  WriteLn('Object now destroyed...');
end;

var
  Ref1, Ref2: IInterface;
begin
  Ref1 := TTest.Create; //ref count now 1
  Ref2 := Ref1;         //ref count now 2
  Ref2 := nil;          //ref count back down to 1
  Ref1 := nil;          //ref count 0 → object freed
  WriteLn('Has the object been destroyed yet?');
end.
```

One benefit of reference counting is that you need not (and in fact, should not) call `Free` on interfaced objects.

Avoiding cyclic references: using 'weak' references

While a relatively simple technique, the reference counting model interfaces support generally works quite well. However, one scenario where it falls down is when an object holds a reference to second that has its own reference to the first (so-called 'cyclic references').

For example, imagine a list object of some sort called `TMyList` that implements the `IMyList` interface. Instances contain items that implement the `IMyItem` interface:

```
type
  IMyItem = interface
    //...
    function GetParent: IMyList;
    property Parent: IMyList read GetParent;
  end.
```

If the `TMyList` class holds references to the items typed to `IMyItem`, and the `IMyItem` implementer(s) hold references to the list typed to `IMyList`, then their instances will keep each other alive.

The solution to this sort of problem is for one side to use so-called 'weak references' to the other. This can be done by using a backing field typed to either `Pointer` or a suitable class, with typecasting used when it needs to be accessed or set as a reference to the actual interface:

```
type
```

```
  TMyItem = class(TInterfacedObject, IMyItem)
  strict private
    FParent: Pointer;
  protected
    function GetParent: IMyList;
  public
    constructor Create(const AParent: IMyList);
    //...
  end.

constructor TMyItem.Create(const AParent: IMyList);
begin
  inherited Create;
  FParent := Pointer(AParent);
end;

function TMyItem.GetParent: IMyList;
begin
  Result := IMyItem(FParent);
end;
```

Interface querying: as, is, hard casts and Supports

In order to go from one interface type to another, you can use the `as` operator. If the object does indeed implement the second interface, the type conversion goes through, otherwise an `EIntfCastError` exception is raised:

```
uses
  System.SysUtils;

type
  ISomethingElse = interface
    ['{3419B526-C616-4DD0-8606-3166C92FBDBE}']
    procedure TestMe;
  end;

var
  Intf: IInterface;
begin
  Intf := TFoo.Create;
  (Intf as IFoo).CallMe;            //fine
  (Intf as ISomethingElse).TestMe; //runtime exception
end.
```

Casting from an interface to a class type also works, using both the 'hard cast' syntax and the `as` operator:

```
procedure Test(const Intf: IMyIntf);
var
  Obj: TMyObj;
begin
  Obj := TMyObj(Intf);    //use the 'hard cast' syntax
  Obj := Intf as TMyObj;  //use the 'as' operator
```

The difference between the two forms is what happens when the query fails (i.e., when `Intf` isn't actually an instance of `TMyObj`). In such a case, where the hard cast will return `nil`, the `as` operator will raise an exception.

The `is` operator can also be used to test for an interface from an existing object:

```
if Intf is TMyObj then TMyObj(Intf).DoSomething;
```

However, you cannot use `is` to query from one interface type to another (`if Intf is IMyOtherIntf`). In its place stand `Supports`, an overloaded utility function declared by `System.SysUtils`:

```
function Supports(const Instance: IInterface; const IID: TGUID;
  out Intf): Boolean; overload;
function Supports(const Instance: TObject; const IID: TGUID;
  out Intf): Boolean; overload;
function Supports(const Instance: IInterface;
  const IID: TGUID): Boolean; overload;
function Supports(const AClass: TClass;
  const IID: TGUID): Boolean; overload;
```

Notwithstanding the fact the IID parameter is typed to `TGUID`, you can pass an interface's name to it, so long as that interface was declared with a GUID (this actually works with *any* parameter typed to `TGUID`):

```
var
  Intf: IInterface;
  SomethingIntf: ISomethingElse;
begin
```

```
  Intf := TTest.Create;
  if Supports(Intf, ISomethingElse, SomethingIntf) then
    SomethingIntf.TestMe
  else
    WriteLn('Doesn''t implement ISomethingElse');
end.
```

In effect, GUIDs plays the role for interfaces here that metaclasses would do in the case of classes.

Interfaces beyond the basics: the IInterface type

Similar to how every Delphi class ultimately inherits from `TObject`, every Delphi interface ultimately derives from `IInterface`. By design, this type is binary compatible with `IUnknown`, the base interface type of the Component Object Model (COM) on Windows. This is not to say Delphi interfaces actually depend on COM, since they don't. Nonetheless, it does mean `IInterface` is composed by the following three methods, exactly like `IUnknown`:

```
function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
function _AddRef: Integer; stdcall;
function _Release: Integer; stdcall;
```

These provide the basic infrastructure for interface querying (i.e., `as` casts and `Supports` calls), together with interface reference counting.

The fact these methods are explicit makes it is possible for an object to not support core interface functionality even when it is only being accessed through an interface reference. In particular, while the language guarantees that `_AddRef` and `_Release` will be called automatically, it does not guarantee that any reference count even exists, let alone that it will be incremented and decremented by those calls.

In principle, the same goes for `QueryInterface`, however if interface querying doesn't work by the book, then it is unlikely to be by design. If you are writing your own interface implementations independent of `TInterfacedObject`, you will therefore usually just do what `TInterfacedObject` does for that method, which is to delegate to the `GetInterface` method defined by `TObject` for the purpose, returning `S_OK` if it succeeds or `E_NOINTERFACE` otherwise:

```
function TFoo.QueryInterface(const IID: TGUID; out Obj): HResult;
begin
  if GetInterface(IID, Obj) then
    Result := S_OK
  else
    Result := E_NOINTERFACE;
end;
```

`_AddRef` and `_Release` are a somewhat different matter, since implementing them to simply return -1 is a common convention for when you don't want reference counting to occur:

```
function TDummyIntfImpl._AddRef: Integer;
begin
  Result := -1;
end;

function TDummyIntfImpl._Release: Integer;
begin
  Result := -1;
end;
```

There is one important caveat about doing this however: the `_AddRef` and `_Release` methods will still be called, which can lead to nasty access violations if you are not careful. This stems from the fact every interface method call is effectively a virtual method call, and virtual methods require valid object references to work against. To avoid the possibility of `_Release` calls being made against stale references, ensure interface references are explicitly `nil`'ed if they will not be going out of scope naturally before the underlying objects will be destroyed.

Interfaces vs. components?

If you can help it, the issue of `_Release` getting called on an already destroyed object shouldn't occur though, since if you use interfaces consistently, there will be no need to disable normal reference counting in the first place. There is however one major exception to this rule, which is if you want to mix components with interfaces, for example have a form implement one or more interfaces (both the VCL and FireMonkey `TForm` ultimately descend from `TComponent`). This is because the `TComponent` ownership pattern does not mix well with `IInterface` reference counting, given both are designed to automatically free an object when it is no longer needed, yet provide this functionality in mutually exclusive ways.

To illustrate this, say the main form in an application has two buttons, Button1 and Button2, the clicking of which various other forms and frames may wish to be notified of. The native `TComponent` way of one component notifying another of something of interest having happened is to use event properties; however, any one event property can only have one listener assigned to it, which won't do in this case. Using interfaces avoids this problem: if an object wishes to be notified of a button being clicked, it implements a suitable interface that represents a click handler, and registers itself with the main form. When a button is clicked, the form then cycles through the registered objects and calls the interface method corresponding to the event.

While `TComponent` does implement `IInterface`, it therefore does so only to provide dummy `_AddRef` and `_Release` methods that do nothing but return -1, as previously demonstrated. Generally, you shouldn't store interface references provided by a component, or if you do, store the references as *component* references, in which case the `TComponent` 'free notification' system can be hooked into (this system was discussed in the previous chapter). Given the sorts of scenario where interfaces are useful with `TComponent` descendants, that need not be much (if any) of a burden however.

Putting words into code, the button click events can have their interfaces defined like this (the interface definitions would all go in their own unit):

```
type
  IButton1ClickedObserver = interface
  ['{7EEE6C3A-D500-4C57-BD6F-27986219E3D8}']
    procedure Button1Clicked;
  end;

  IButton2ClickedObserver = interface
  ['{F83330BF-49B1-4549-9C7E-D279A19D3778}']
    procedure Button2Clicked;
  end;
```

The main form can also have its own interface defined *qua* subject of the events:

```
  IAppEventsSubject = interface
  ['{0EEBBCCB-968E-4CE9-AC13-AAB4D342269F}']
    procedure RegisterObserver(Observer: TComponent);
  end;
```

Forms and frames that wish to be notified of the click events will then implement one or both of `IButton1ClickedObserver` and `IButton2ClickedObserver`, before calling the `RegisterObserver` method to register themselves when first created.

When only forms are involved and the event subject is guaranteed to be the main form, the observers could just query for `IAppEventsSubject` from the `MainForm` property of the `Application` global object:

```
procedure TfrmBtn1Observer.FormCreate(Sender: TObject);
var
  Obj: IAppEventsSubject;
begin
  if Supports(Application.MainForm, IAppEventsSubject, Obj) then
    Obj.RegisterObserver(Self);
end;
```

Alternatively, the unit that defines the event interfaces might manage an explicit reference to the `IAppEventsSubject` implementer, which the main form can set on its creation and clear on its destruction. Either way, you should avoid the observer units directly using the main form's unit if you can.

In implementing `RegisterObserver`, a useful class to employ is `TComponentList`, which is declared in `System.Contnrs`. This implements a list of `TComponent` instances, as its name implies, but hooks into the `FreeNotification` system to automatically remove items from the list as they are freed. This avoids the possibility of `_Release` being called on a stale reference, and removes the need for an explicit `UnregisterObserver` method that clients need to remember to call:

```
uses
  {...}, System.Contnrs;

type
  TfrmMain = class(TForm, IAppEventsSubject)
  //...
  strict private
    FObservers: TComponentList;
    procedure RegisterObserver(Observer: TComponent);
  //...
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  //...

implementation
```

```
constructor TfrmMain.Create(AOwner: TComponent);
begin
  FObservers := TComponentList.Create;
  inherited;
end;

destructor TfrmMain.Destroy;
begin
  FObservers.Free;
  inherited;
end;

procedure TfrmMain.RegisterObserver(Observer: TComponent);
begin
  if (Observer <> nil) and (FObservers.IndexOf(Observer) < 0) then
    FObservers.Add(Observer);
end;
```

To actually notify registered observers of an event, the `FObservers` list is enumerated, implementers of the relevant interface picked out, and the interface method called:

```
procedure TfrmMain.Button1Click(Sender: TObject);
var
  I: Integer;
  Obj: IButton1ClickedObserver;
begin
  for I := 0 to FObservers.Count - 1 do
    if Supports(FObservers[I], IButton1ClickedObserver, Obj) then
      Obj.Button1Clicked;
end;
```

Interface aggregation: the 'implements' keyword

There may be occasions in which an object should formally implement a number of interfaces whose actual implementation is elsewhere. For example, it might be desirable to share a particular implementation of an interface between multiple classes. Alternatively, it may seem sensible to break down a class that implements many interfaces into more manageable chunks; delegating the implementation of individual parts to individual helper classes would allow this to happen while still preserving the class's singular identity to the outside world.

To illustrate, imagine the following two interfaces, the first of which the main object implements directly and the second it wishes to delegate to a helper class:

```
type
  IMainIntf = interface
  ['{864B80CB-AE6D-46D2-896D-4A989D09F008}']
    procedure Foo;
  end;

  ISubIntf = interface
  ['{151CC153-8CDA-4A6F-AF6D-7C777019ABF8}']
    procedure SayHello;
    procedure SayGoodbye;
  end;
```

The helper class itself is then defined like this:

```
type
  TSubObj = class
  public
    procedure SayHello;
    procedure SayGoodbye;
  end;
```

The naïve way for the main object to defer `ISubIntf`'s implementation to a `TSubObj` instance would be to explicitly delegate on a method-by-method basis:

```
type
  TMainObj = class(TInterfacedObject, IMainIntf, ISubIntf)
  strict private
    FSubObj: TSubObj;
  protected
    { IMainIntf - implemented ourselves }
    procedure Foo;
    { ISubIntf - delegated to FSubObj }
    procedure SayHello;
    procedure SayGoodbye;
  public
    constructor Create;
    destructor Destroy; override;
  end;

constructor TMainObj.Create;
begin
  inherited Create;
  FSubObj := TSubObj.Create;
end;

destructor TMainObj.Destroy;
begin
  FSubObj.Free;
  inherited Destroy;
end;

procedure TMainObj.SayHello;
begin
  FSubObj.SayHello;
end;

procedure TMainObj.SayGoodbye;
begin
  FSubObj.SayGoodbye;
end;
```

While doing such a thing is tolerable in simple cases, it gets tedious in more complex ones where there are lots of methods to delegate. Because of this, the Delphi language provides the `implements` keyword to automate interface delegation. This is used in the context of a property declaration:

```
type
  TMainObj = class(TInterfacedObject, IMainIntf, ISubIntf)
  strict private
    FSubObj: TSubObj;
  protected
    { IMainIntf - implemented ourselves }
    procedure Foo;
    { ISubIntf - delegated to FSubObj }
    property SubObj: TSubObj read FSubObj implements ISubIntf;
  public
    constructor Create;
    destructor Destroy; override;
  end;
```

When the `implements` keyword is used, the compiler looks for method implementations in the sub-object. If the sub-object can provide all the methods for the interface specified, then great, otherwise the remainder will fall back on the container object to implement directly.

In fact, the syntax is flexible enough for the container to directly implement a method even if the sub-object it has delegated the interface's implementation to generally has the required method: just use the method resolution syntax we met earlier:

```
type
  TMainObj = class(TInterfacedObject, IMainIntf, ISubIntf)
  strict private
    FSubObj: TSubObj;
  protected
    property SubObj: TSubObj read FSubObj implements ISubIntf;
    procedure DoSayHello;
    { avoid SubObj's implementation of SayHello }
    procedure ISubIntf.SayHello = DoSayHello;
```

Aggregation hassles

While the `implements` keyword is pretty slick, it unfortunately has its pitfalls too. If you look carefully at the code just presented, you will see `TSubObj` was not declared as implementing any interfaces itself. For sure, it contained the `ISubIntf` methods, but it didn't actually support `ISubIntf` in the formal sense. In fact, by descending straight from `TObject`, it didn't even informally implement `ISubIntf` *in toto*. This is because `ISubIntf` (as every Delphi interface) implicitly extends `IInterface`, and `TSubObj` (being a direct descendant of `TObject`) did not contain the `IInterface` methods (`QueryInterface`, `_AddRef` and `_Release`).

This point can be verified if the declaration of `TSubObj` is amended to say it *does* implement `ISubIntf`:

```
type
  TSubObj = class(TObject, ISubIntf)
  public
    procedure SayHello;
    procedure SubGoodbye;
  end;
```

Try this, and you'll find the compiler complaining about not being able to find the `IInterface` methods. So, descend from `TInterfacedObject` to ensure the magic three methods are there:

```
type
  TSubObj = class(TInterfacedObject, ISubIntf)
  public
    procedure SayHello;
    procedure SubGoodbye;
  end;
```

Next, add the following test code:

```
var
  Intf: IInterface;
  MainIntf: IMainIntf;
  SubIntf: ISubIntf;
begin
  try
    Intf := TMainObj.Create;
    SubIntf := Intf as ISubIntf;
    MainIntf := SubIntf as IMainIntf;
    WriteLn('Worked!');
  except
    on E: Exception do
      WriteLn(E.ClassName, ': ', E.Message);
  end;
  ReadLn;
end.
```

Run this through the debugger, and you will find an exception getting raised, after which the application will hang, notwithstanding the `try/except` block. Nasty! If you were to use exactly the same test code with the original version though (i.e., where `TSubObj` descended from `TObject` directly), it will run fine. Why so?

In a nutshell, because the newly-introduced `IInterface` implementation of `TSubObj` conflicts with the `IInterface` implementation of the parent object. If we take the test code step by step, the first `as` cast calls the `QueryInterface` method provided by `TMainObj`, via its parent class (`TInterfacedObject`). Since the second `as` cast is on the `ISubIntf` reference however, it matters where the `QueryInterface` method of `ISubIntf` is implemented. When `TSubObj` descended directly from `TObject`, it provided no `QueryInterface` implementation itself, in which case `TMainObj`'s implementation was picked up by the compiler (which is what we wanted!). In contrast, when `TSubObj` descended from `TInterfacedObject`, it *did* have a `QueryInterface` method to be used.

This explains the casting exception, since `TSubObj` only implemented `ISubObj`, and the second cast was effectively querying from a `TSubObj` to an `IMainIntf`. The fact the program then hung has a similar explanation: when `TSubObj` descends from `TInterfacedObject`, it gets its own `_AddRef` and `_Release` methods, along with its own reference count. The `TMainObj` instance, however, continued to assume it was responsible for the sub-object's disposal. Given the first `as` cast in the test code was successful, this lead to both `TMainObj` and `TSubObj` attempting to destroy the same instance. Oh dear...

The COM aggregation pattern

The solution to this problem comes in two main forms: either ensure the sub-object does not provide any `IInterface` methods itself, which can be done by having its class directly descend from `TObject`, or have its `IInterface` implementation written in such a way that requests for incrementing or decrementing the reference count, or querying for other interfaces, are passed onto the main ('controller') object as appropriate.

This second option is sometimes called the 'COM aggregation pattern'. Implementing it can be more or less involved, though in Delphi, the simplest way is to descend from `TAggregatedObject` rather than (say) `TInterfacedObject`:

```
type
  TSubObj = class(TAggregatedObject, ISubIntf)
  public
    procedure SayHello;
    procedure SubGoodbye;
  end;
```

In practice, if you're free to amend the sub-object's base class in this way, you would be free to descend it from `TObject` directly, and so not have aggregation hassles in the first place.

More likely is a situation in which an object may implement a particular interface *either* for its own sake *or* for the sake of a container. In that case, you need to reimplement `QueryInterface`, `_AddRef` and `_Release` so that they get passed on to the container (held via a ‘weak reference’) if aggregation is in effect:

```
type
  TSubObj = class(TInterfacedObject, IInterface, ISubIntf)
  strict private
    { hold only a weak reference to the outer object }
    FController: Pointer;
  protected
    { must redefine the IInterface methods when aggregated }
    function QueryInterface(const IID: TGUID;
      out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
    { ISubIntf’s own methods as before }
    procedure SayHello;
    procedure SubGoodbye;
  public
    constructor CreateAggregated(const AController: IInterface);
  end;

constructor TSubObj.CreateAggregated(const AController: IInterface);
begin
  inherited Create;
  FController := Pointer(AController);
end;

function TSubObj.QueryInterface(const IID: TGUID; out Obj): HResult;
begin
  if FController <> nil then
    Result := IInterface(FController).QueryInterface(IID, Obj)
  else
    Result := inherited QueryInterface(IID, Obj);
end;

function TSubObj._AddRef: Integer;
begin
  if FController <> nil then
    Result := IInterface(FController)._AddRef
  else
    Result := inherited _AddRef;
end;

function TSubObj._Release: Integer;
begin
  if FController <> nil then
    Result := IInterface(FController)._Release
  else
    Result := inherited _Release;
end;
```

In its constructor, `TMainObj` would now replace the call to `TSubObj.Create` with one to `TSubObj.CreateAggregated`, passing `Self` as the parameter. `TMainObj.Destroy` will however remain as it was originally, with the sub-object being explicitly freed, since when aggregated the object will not have its own independent reference count.

Generics

A ‘generic’ is a sort of blueprint for a class, record, interface or method that takes one or more type parameters. The following generic record, for example, defines a single type parameter called `T` (the name itself is arbitrary), which a field called `Value` is typed to:

```
type
  TMyGeneric<T> = record
    Value: T;
  end;
```

The definition of `T` here is ‘unconstrained’. This means it can be filled out — the generic record ‘instantiated’ — with any type whatsoever:

```
var
  MyStrInst: TMyGeneric<string>;
  MyIntInst: TMyGeneric<Integer>;
  MyEncodingInst: TMyGeneric<TEncoding>;
begin
  MyStrInst.Value := 'Hello Generics World!';
  MyIntInst.Value := 42;
  MyEncodingInst.Value := TEncoding.Default;
  if MyEncodingInst.Value.IsSingleByte then //...
```

In each case, the record’s `Value` field is typed to what is put between the angle brackets when the generic is instantiated. Consequently, a compiler error will arise if you attempt to assign a string to `MyIntInst.Value`, an integer to `MyStrInst.Value`, and so forth.

A popular use of generics is when defining container types, for example lists of such-and-so. Here, the such-and-so is parameterised so that the same list class code can be used for lots of different item types — lists of strings, lists of integers, lists of `TMyObject`, and so on. In the Delphi RTL, the `System.Generics.Collections` unit provides numerous generic classes in this vein. These will be looked at in detail in chapter 6. In this section, in contrast, we will look at generics purely in terms of the Delphi language.

Sorts of generic

The syntax for a generic class or interface is very similar to a generic record:

```
type
  TMyGenericObj<T> = class
    Value: T;
  end;

  IMyGenericIntf<T> = interface
    function GetValue: T;
  end;
```

In general, you should be careful with generic interfaces though, since all instantiations will share the same GUID (if any).

Aside from generic types, both records and classes (but not interfaces) can have generic methods, regardless of whether the wider type is itself generic:

```
type
  TTest = record
    procedure Foo<T>(Param: T);
    procedure Foo2<T>(Param: T);
  end;
```

In this example, `Foo`’s `T` is completely independent from `Foo2`’s. When used, generic methods support type inference in simple cases. This means the type parameter doesn’t have to be explicitly filled out:

```
var
  Rec: TTest;
begin
  Rec.Foo<string>('works'); //instantiate explicitly
  Rec.Foo('also works');    //use type inference
```

Aside from generic instance methods, generic class and static methods are supported too. Generic global routines are *not* however. Nevertheless, generic static methods form a reasonable substitute for them:

```
type
  TMyGlobals = record
    class procedure Foo<T>(const Value: T); static;
    class function ReturnMe<T>(const Value: T): T; static;
  end;
```

```
class procedure TMyGlobals.Foo<T>(const Value: T);
begin
  //do something...
end;

class function TMyGlobals.ReturnMe<T>(const Value: T): T;
begin
  Result := Value;
end;
```

In use:

```
var
  S: string;
begin
  TMyGlobals.Foo<Integer>(42);
  S := TMyGlobals.ReturnMe('Hello world!');
```

Defining constraints

When a parameterised type is left unconstrained, the generic can only perform operations on it that can be performed on the instances of *any* type. Consequently, the following code raises a compiler error since only certain types support the + operator:

```
type
  TTest<T> = class
    class function Add(const P1, P2: T): T;
  end;

class function TTest<T>.Add(const P1, P2: T): T;
begin
  Result := P1 + P2;
end;
```

Sometimes this need not be a problem; for example, the stock list and list-type classes provided by the `System.Generics.Collections` unit are mostly unconstrained. However, in other situations, constraints can be necessary if a generic is to do anything useful.

To apply a constraint to a type parameter means requiring it to be instantiated with certain sorts of type only. For example, you might require instantiating with a class type:

```
type
  TConstrainedRec<T: class> = record
    Value: T;
    procedure WorkWithValue;
  end;

  TTestObject = class
  end;

  TTestRec = record
  end;

var
  Valid: TConstrainedRec<TTestObject>;   //compiles
  Invalid: TConstrainedRec<TTestRecord>; //doesn't compile
```

Once a type parameter is constrained, the generic's implementation can use any method, standard routine or operator supported by all instances of the type kind constrained to:

```
procedure TConstrainedRec<T>.WorkWithValue;
begin
  WriteLn(Value.ClassName); //wouldn't compile if unconstrained
end;
```

By their very nature, using constraints well means finding a suitable equilibrium. Constrain too much, and the generic won't be very 'generic'; constrain too little though, and it won't be able to do very much.

Constraint syntax

The syntax for generic constraints is similar to the syntax for declaring an ordinary parameter list, in that the name of each type parameter is followed by a colon then the type kinds it should be constrained to:

```
type
  IFoo = interface
  ['{7BAF1ACB-3CC6-4D59-8F2F-C169BBE30609}']
    //...
  end;

  TBar = class
    //...
  end;

  TTestRec<T1: TBar; T2: class; T3: record; T4: IFoo> = record
    //...
  end;
```

In this example, the first type parameter must be either `TBar` or a descendant class of `TBar`, the second any class whatsoever, the third a value type (not just a record!), and the fourth either a class that implements `IFoo`, the `IFoo` interface itself, or an interface that extends `IFoo`:

```
type
  TMyRecord = record
    //...
  end;

  TFoo = class(TInterfacedObject, IFoo)
    //...
  end;

  IFooEx = interface(IFoo)
    procedure ExtraMethod;
  end;

  TBarman = class
    //...
  end;

  TNightclub = class(TBar)
    //...
  end;

  TBouncer = class
    //...
  end;

  TSomeEnum = (None, First, Second, Third);

var
  MyInst1: TFoo<TBar, TBarman, TMyRecord, TFoo>;
  MyInst2: TFoo<TNightclub, TBouncer, TSomeEnum, IFooEx>;
```

With the exception of value type constraints, which must be used alone, each type parameter can receive more than one constraint. Multiple constraints work on a logical AND basis, like the more demanding clients of an estate agent: only a house with a big garden *and* close to the centre of town will do!

As declared, multiple constraints must be separated from each other with a comma:

```
type
  TMultiTest<MyType: TComponent, IStreamPersist> = class
    procedure RenameAndStreamData(Value: MyType;
      const NewName: string; OutStream: TStream);
  end;
```

Here, `MyType` must be a `TComponent` descendant that implements `IStreamPersist`. Because of this double constraint though, `RenameAndStreamData` can both set properties declared on `TComponent` and call methods defined by `IStreamPersist`:

```
procedure TMultiTest<MyType>.RenameAndStreamData(Value: MyType;
  const NewName: string; OutStream: TStream);
begin
  Value.Name := NewName;         //set a property of TComponent
  Value.SaveToStream(OutStream); //call method of IStreamPersist
end;
```

True strong typing: no type erasure

Originally, Delphi did not support generics. Because of that, the basic list class provided by the original RTL — `TList`, now of `System.Classes` — was a list of untyped pointers. This class required lots of potentially unsafe typecasts in use:

```
uses System.Classes;

var
```

```
  FButtons: TList;

//...

var
  I: Integer;
begin
  for I := 0 to FButtons.Count - 1 do
    { Is FButtons[I] REALLY a TButton? }
    if TButton(FButtons[I]).Visible then
      TButton(FButtons[I]).Enabled := True;
```

For the sake of backwards compatibility, the old `TList` class still exists. Alongside it stands a newer, generics-enabled version though:

```
uses System.Generics.Collections;

var
  FButtons: TList<TButton>;

//...

var
  I: Integer;
begin
  for I := 0 to FButtons.Count - 1 do
    if FButtons[I].Visible then
      FButtons[I].Enabled := True;
```

One approach to generics — an approach adopted by the Java language — is to employ a process of 'type erasure', in each the compiler resolves every instantiation of a generic to a non-generic underlying type. In the `TList` example, that would mean `TList<TButton>`, along with `TList<string>` and `TList<WhateverType>`, resolving down to either the classic, non-generic `TList` or something very similar.

Delphi generics, however, are not like that. Fundamentally, `TList<TButton>` and `TList<string>` are completely separate classes, with separate inheritance trees that the compiler backfills as necessary. So, as declared, the generic `TList` inherits from a generic `TEnumerable` class, which itself inherits from `TObject`. The parent class of `TList<TButton>` is therefore `TEnumerable<TButton>`, which descends directly from `TObject`. Likewise, the parent class of `TList<string>` is `TEnumerable<string>`, which descends directly from `TObject` too. The following code can be used to verify this:

```
uses
  System.Classes, System.Generics.Collections;

procedure OutputClassHierarchy(ClassType: TClass);
var
  Spacing: string;
begin
  Spacing := '';
  while ClassType <> nil do
  begin
    WriteLn(Spacing + ClassType.ClassName);
    ClassType := ClassType.ClassParent;
    Spacing := Spacing + '  ';
  end;
end;

var
  OldList: TList;
  IntList: TList<Integer>;
  StrList: TList<string>;
begin
  OutputClassHierarchy(TList);
  OutputClassHierarchy(TList<Integer>);
  OutputClassHierarchy(TList<string>);
end.
```

This outputs the following:

```
TList
  TObject
TList<System.Integer>
  TEnumerable<System.Integer>
    TObject
TList<System.string>
  TEnumerable<System.string>
    TObject
```

Generics and polymorphism

The fact type erasure is not used by the Delphi compiler makes it impossible to naively pass an instantiation of a generic type to a routine that (for historical or other reasons) expects a non-generic type:

```
uses System.SysUtils, System.Classes, System.Generics.Collections;

procedure WorkWithLegacyList(List: TList);
begin
  { Do stuff, possibly on the basis that List is a list of
    TEncoding, possibly not... }
end;

procedure TestGeneric;
var
  List: TList<Integer>;
begin
  List := TList<Integer>.Create;
  WorkWithLegacyList(List); //compiler error!
end;
```

This is generally a good thing, because it prevents you from mistakenly passing a list of (say) strings to a routine that expects a list of integers. Furthermore, it is also slightly more performant that the type erasure approach, since explicit casts have not just been replaced with implicit ones. However, it can also be frustrating. For example, the following doesn't compile:

```
uses System.Generics.Collections;

type
  TAnimal = class
    procedure MakeSound; virtual;
  end;

  TCat = class(TAnimal)
    procedure MakeSound; override;
  end;

  TDog = class(TAnimal)
    procedure MakeSound; override;
  end;

procedure MakeNoise(List: TList<TAnimal>);
var
  Animal: TAnimal;
begin
  for Animal in List do
    Animal.MakeSound;
end;

var
  CatList: TList<TCat>;
begin
  //...
  MakeNoise(CatList); //compiler error!
```

The problem here is that while `TCat` is a descendant of `TAnimal`, `TList<TCat>` is *not* a descendant of `TList<TAnimal>`: rather, it is a child of `TEnumerable<TCat>`, which is a direct descendant of `TObject`. If you were allowed to pass a `TList<TCat>` where a `TList<TAnimal>` is expected, the following sort of code would compile:

```
procedure DoWork(Data: TObjectList<TAnimal>);
begin
  Data.Add(TDog.Create);
end;

var
  CatList: TList<TCat>;
begin
  //...
  DoWork(CatList);
```

The compiler is plainly doing its job when it disallows adding a `TDog` to a list of `TCat` though.

As it stands, `MakeNoise` therefore simply can't accept a `TList<TCat>`. The workaround is to make `MakeNoise` *itself* generic, with a type parameter that will correspond to the type parameter of the list class passed in:

```
type
```

```
  TAnimalUtils = record
    class procedure MakeNoise<T: TAnimal>(List: TList<T>); static;
  end;

class procedure TAnimalUtils.MakeNoise<T>(List: TList<T>);
var
  Animal: TAnimal;
begin
  for Animal in List do
    Animal.MakeSound;
end;
```

The call to `MakeNoise` then needs to be amended accordingly:

```
  TAnimalUtils.MakeNoise<TCat>(List);
```

The moral here is that a generic must be 'closed off' as late as possible in order to maintain flexibility.

Constraining constraints

As previously noted, if a generic type parameter is to be useful, it frequently needs to be constrained. Unfortunately, the range of possible constraints is very limited however. For example, it is not possible to constrain to types that support one or more operators, or that a given method defined, or is a 'number', or is an ordinal type, and so on.

In the operator case, one possible workaround is to use a helper type. For example, imagine writing a generic maths library. For simplicity, let's make it a single type with a single static method, `Sum`:

```
type
  TGenericMaths<T> = record
    class function Sum(const ANumbers: array of T): T; static;
  end;
```

As defined, we can't actually implement `Sum`, since we would need to use the addition operator, and we can't constrain to types supporting certain operators. What `TGenericMaths` needs is therefore some sort of helper type to do the addition (and potentially other primitive operations) for it.

Making the helper type a metaclass with a virtual `Add` method is probably as elegant as this approach can get:

```
type
  TCalculator<T> = class abstract
    class function Add(const A, B: T): T; virtual; abstract;
  end;

  TGenericMaths<T; Calculator: TCalculator<T>> = record
    class function Sum(const ANumbers: array of T): T; static;
  end;

class function TGenericMaths<T, Calculator>.Sum(
  const ANumbers: array of T): T;
var
  Elem: T;
begin
  Result := Default(T); //the default value of a number will be 0
  for Elem in ANumbers do
    Result := Calculator.Add(Result, Elem);
end;
```

For each type `TGenericMaths` may be instantiated with, a corresponding `TCalculator` helper class now needs to be defined and 'plugged in' at the same time `TGenericMaths` itself is used:

```
type
  TIntegerCalculator = class(TCalculator<Integer>)
    class function Add(const A, B: Integer): Integer; override;
  end;

  TDoubleCalculator = class(TCalculator<Double>)
    class function Add(const A, B: Double): Double; override;
  end;

  TSingleCalculator = class(TCalculator<Single>)
    class function Add(const A, B: Single): Single; override;
  end;

class function TIntegerCalculator.Add(const A, B: Integer): Integer;
begin
  Result := A + B;
end;
```

```
class function TDoubleCalculator.Add(const A, B: Double): Double;
begin
  Result := A + B;
end;

class function TSingleCalculator.Add(const A, B: Single): Single;
begin
  Result := A + B;
end;
```

The generic `Sum` method can now be used like this:

```
var
  IntegerArray: array[1..100000] of Integer;
  SingleArray: array[1..100000] of Single;
  DoubleArray: array[1..100000] of Double;
  IntTotal: Integer; SnlTotal: Single; DblTotal: Double;
begin
  //...
  IntTotal := TGenericMaths<Integer,TIntegerCalculator>.Sum(
    IntegerArray);
  DblTotal := TGenericMaths<Double,TDoubleCalculator>.Sum(
    DoubleArray);
  SnlTotal := TGenericMaths<Single,TSingleCalculator>.Sum(
    SingleArray);
```

The constructor constraint

When a class constraint is used, a common scenario is for the generic to instantiate the class parameter at some point:

```
type
  TSingleton<T: class> = record
  strict private
    class var FInstance: T;
    class function GetInstance: T; static;
  public
    class property Instance: T read GetInstance;
  end;

class function TSingleton<T>.GetInstance: T;
begin
  if FInstance = nil then FInstance := T.Create;
  Result := FInstance;
end;
```

Simple though this code is, it will not compile. Instead, it will cause an error complaining about a missing `constructor` constraint. Fixing this immediate problem is easy enough:

```
type
  TSingleton<T: class, constructor> = record
  //...
```

This allows the `GetInstance` method to now compile. What happens if the class used to instantiate the generic doesn't define a parameterless constructor called `Create` though? If you were to try constructing such a class directly, it wouldn't work unless the custom constructor is called:

```
type
  TTestObject = class
  strict private
    FTitle: string;
  public
    constructor Create(const ATitle: string);
    property Title: string read FTitle;
  end;

var
  Test: TTestObject;
begin
  Test := TTestObject.Create; //compiler error!
```

However, through a generic it *does* compile, though at the cost of any custom constructor not running. The following therefore outputs a blank line:

```
WriteLn(TSingleton<TTestObject>.Instance.Title);
```

The fact the generic case compiles is because `TObject` (the ultimate base class of all Delphi classes) defines a parameterless constructor called `Create`. As soon as you define a constructor of your own, the default one is 'hidden' in non-generic code. However, since a generic `class` constraint constrains to all classes, what the generic 'see' are the `TObject` methods... including the default constructor. The fix is to provide a parameterless constructor called `Create` yourself:

```
type
  TTestObject2 = class
  strict private
    FTitle: string;
  public
    constructor Create;
    property Title: string read FTitle;
  end;

constructor TTestObject2.Create;
begin
  FTitle := 'OK';
end;

begin
  WriteLn(TSingleton<TTestObject2>.Instance.Title);
```

This now outputs `OK`, as you would expect.

Beyond the constructor constraint

Unfortunately, the `constructor` constraint only allows constructing a class using a parameterless `Create` constructor. The following therefore ends in a compiler error:

```
type
  TBaseObject = class abstract
    constructor Create(const ATitle: string); virtual; abstract;
  end;

  TSingleton<T: TBaseObject, constructor> = record
  strict private
    class var FInstance: T;
    class function GetInstance: T; static;
  public
    class property Instance: T read GetInstance;
  end;

class function TSingleton<T>.GetInstance: T;
begin
  if FInstance = nil then
    FInstance := T.Create('Test'); //compiler error!
  Result := FInstance;
end;
```

Luckily, the workaround is pretty trivial — make the metaclass of the class being constrained to explicit, and use a couple of typecasts when constructing the object:

```
type
  TBaseObject = class abstract
    constructor Create(const ATitle: string); virtual; abstract;
  end;

  TBaseClass = class of TBaseObject;

class function TSingleton<T>.GetInstance: T;
var
  LClass: TBaseClass;
begin
  if FInstance = nil then
    FInstance := Pointer(TBaseClass(T).Create('Test'));
  Result := FInstance;
end;
```

Neither typecast should really be necessary, but they are, and the fact isn't worth worrying about.

Anonymous methods

Voltaire, the French Enlightenment philosopher, quipped that the Holy Roman Empire was neither holy, nor Roman, nor an empire. While it would be an exaggeration to say anonymous methods are neither anonymous nor methods, when they are anonymous, they don't seem to be methods, and when are methods, they don't seem to be anonymous.

In a nutshell, an anonymous method is a procedure or function that is embedded inside another procedure or function, can be assigned to a variable, and can carry state in its capacity as a 'closure'. Anonymous 'methods', therefore, are not obviously attached to any object, despite their name. On the other hand, a variable typed to an anonymous method can be assigned an ordinary method too, in which case the variable will point to a method that isn't anonymous! As we will be seeing, this turns out to be the least of anonymous methods' tricks though.

Basic syntax

As defined, an anonymous method type (also called a 'method reference' type) looks similar to an event or 'method pointer' type, only with the `of object` suffix replaced with a `reference to` prefix:

```
type
  TMyAnonMethod = reference to procedure (SomeParam: Integer);
```

`TMyAnonMethod` will be type compatible with an anonymous method that takes an `Integer` parameter, an ordinary method that takes one, or a standalone procedure that does so:

```
procedure Standalone(V: Integer);
begin
  WriteLn('You passed ', V, ' to the standalone procedure');
end;

type
  TTest = class
    procedure OrdinaryMethod(V: Integer);
  end;

procedure TTest.OrdinaryMethod(V: Integer);
begin
  WriteLn('You passed ', V, ' to the ordinary method');
end;

procedure TestAssignment;
var
  Ref1, Ref2, Ref3: TMyAnonMethod;
  Obj: TTest;
begin
  Ref1 := Standalone;
  Obj := TTest.Create;
  Ref2 := Obj.OrdinaryMethod;
  Ref3 := procedure(V: Integer)
          begin
            WriteLn('You passed ', V, ' to the anon method');
          end;
  //invoke the references just assigned
  Ref1(1);
  Ref2(2);
  Ref3(3);
end;
```

As shown here, the syntax of an anonymous method is identical to that for an ordinary procedure or function, only being nameless and without a semi-colon immediately after the header.

Predefined method reference types

For convenience, the `System.SysUtils` unit defines a group of generic method reference types for you:

```
type
  TProc = reference to procedure;
  TProc<T> = reference to procedure(Arg1: T);
  TProc<T1, T2> = reference to procedure(Arg1: T1; Arg2: T2);
  TProc<T1, T2, T3> = reference to procedure(Arg1: T1; Arg2: T2; Arg3: T3);
  TProc<T1, T2, T3, T4> = reference to procedure(Arg1: T1; Arg2: T2; Arg3: T3; Arg4: T4);

  TFunc<TResult> = reference to function: TResult;
  TFunc<T, TResult> = reference to function(Arg1: T): TResult;
  TFunc<T1, T2, TResult> = reference to function(Arg1: T1; Arg2: T2): TResult;
  TFunc<T1, T2, T3, TResult> = reference to function(Arg1: T1; Arg2: T2; Arg3: T3): TResult;
  TFunc<T1, T2, T3, T4, TResult> = reference to function(Arg1: T1; Arg2: T2; Arg3: T3; Arg4: T4): TResult;

  TPredicate<T> = reference to function(Arg1: T): Boolean;
```

So, if want a type for assigning a method that takes a single `string` argument and returns an `Integer`, you could use the following syntax:

```
var
  MyMethod: TFunc<string, Integer>;
```

That said, the fact predefined types exist shouldn't stop you from defining your own. In some cases indeed, using custom type names can improve code clarity.

Method references as a more flexible sort of callback type

One basic use of method references is to provide a more convenient and flexible form of callback type than that afforded by Delphi's traditional procedural and method pointer (event) types. In the RTL, an example of them being used in this way is with the stock collection classes. Here, an anonymous method can be used to provide a custom comparer for a given type:

```
{ Sort numbers in descending order }
List.Sort(TComparer<Integer>.Construct(
  function (const L, R: Integer): Integer
  begin
    Result := R - L;
  end));
```

When the callback implementation is small, being able to define it 'in place' makes for neater code. Nonetheless, the fact a method reference can be assigned to both an ordinary method and a standalone routine too means you can always separate out the callback implementation if and when you feel it has become too unwieldy to be written in the anonymous fashion.

Referencing variables, not copying values

An anonymous method can freely use variables defined by the outer scope, for example the parameters and local variables of the routine that creates it. Confusing to many people at first is the fact variables are *referenced* rather than their values *copied*. Consider the following code:

```
type
  TSimpleAnonProc = reference to procedure;

var
  Procs: array[1..5] of TSimpleAnonProc;
  Val: Integer;
begin
  for Val := 1 to 5 do
    Procs[Val] :=
      procedure
      begin
        WriteLn(Val);
      end;
  for Val := 1 to 5 do
    Procs[Val]();
end.
```

As you would expect, this outputs 1, 2, 3, 4 and 5. However, now consider this version, the only difference being the way the methods are enumerated on being invoked:

```
var
  Proc: TSimpleAnonProc;
  Procs: array[1..5] of TSimpleAnonProc;
  Val: Integer;
begin
  for Val := 1 to 5 do
    Procs[Val] :=
      procedure
      begin
        WriteLn(Val);
      end;
  for Proc in Procs do
    Proc;
end.
```

In this case, the output is likely to be five 6s. Why so? Because after a successfully completed `for`/`to` loop, the compiler usually leaves the value of the indexer one position beyond the loop's upper bound. Since when being created, the anonymous methods merely referenced `Val`, what gets output at the end is therefore the now-current value of `Val`, i.e. 6. That the original version of the code 'worked' was because the initial incrementing of `Val` was being repeated, setting `Val` to what it was before each anonymous method was fired.

Variable capturing

In the examples just given, the anonymous methods were invoked in the same routine that created them. Since an anonymous method on creation is assigned to a variable, and this variable might be passed to somewhere outside of the creating routine - in fact, it might even be from outside in the first place - an possible problem arises: what happens when a variable or parameter of the originating scope is referred to when that scope no longer exists?

```
uses System.SysUtils;

var
  AnonProc: TProc;

procedure Foo;
var
  FooLocal: string;
begin
  FooLocal := 'I am local to Foo';
  AnonProc := procedure
              begin
                WriteLn(FooLocal);
              end;
end;

begin
  Foo;
  AnonProc; //is this safe?
end.
```

The short answer is that there is nothing to worry about, since in such a case, the anonymous method will preserve any variable or parameter from the originating routine it references, 'hoisting' the item from its original location to ensure it lives for as long as the method itself. If more than one anonymous method references the same outer variable, then they share it:

```
var
  AnonProc1, AnonProc2: TProc;

procedure Foo;
var
  FooLocal: string;
begin
  FooLocal := 'I am local to Foo';
  AnonProc1 := procedure
               begin
                 WriteLn(FooLocal);
                 FooLocal := 'Oh no you''re not!';
               end;
  AnonProc2 := procedure
               begin
                 WriteLn(FooLocal);
               end;
end;

begin
  Foo;
  AnonProc1; //I am local to Foo
  AnonProc2; //Oh no you're not!
end.
```

This process of preserving bits of the originating scope is called variable 'capturing', which has the effect of creating a 'closure'. I confess finding the terminology of 'closures' a bit confusing myself. Saying the variables of the outer scope are 'closed over' by the anonymous method sounds like saying my kitchen has encircled my house - an idea nonsense if taken literally, and still obscure even when taken metaphorically! The basic point, though, is that an anonymous method can safely reference any variable or parameter of its parent routine, regardless of whether the method is invoked long after that routine has finished.

In effect, captured items form an anonymous method's 'state', since they become both private to the capturer and keep their value between calls. This makes captured variables analogous to the fields of an object, which are normally made private and have values that persist between ordinary method calls. In the (rarer) case of a variable that is jointly captured by two or more anonymous methods, the situation is analogous to class variables, i.e. fields that are shared between all instances of the same class.

Constructor functions

While you can create anonymous methods in the middle of one big routine, a common practice is to define specialist 'constructor' functions:

```
function CreateAutoInc(StartFrom: Integer = 0): TFunc<Integer>;
begin
  Result := function : Integer
            begin
              Inc(StartFrom);
              Result := StartFrom;
            end;
end;

var
  AutoInc: TFunc<Integer>;
begin
  Counter := CreateAutoInc(99);
  WriteLn(AutoInc); //100
  WriteLn(AutoInc); //101
  WriteLn(AutoInc); //102
  WriteLn(AutoInc); //103
end.
```

Defining a constructor function like this is also the solution for avoiding the (sometimes) undesirable referencing behaviour of anonymous methods that we met earlier:

```
function CreateSimpleAnonProc(I: Integer): TSimpleAnonProc;
begin
  Result := procedure
            begin
              WriteLn(I);
            end;
end;

var
  Proc: TSimpleAnonProc;
  Procs: array[1..5] of TSimpleAnonProc;
  I: Integer;
begin
  for I := 1 to 5 do
    Procs[I] := CreateSimpleAnonProc(I);
  for Proc in Procs do
    Proc;
end.
```

One way to look at these constructor functions is to consider them factories, manufacturing versions of a given function or procedure according to particular requirements:

```
type
  TConverterFunc = reference to function (const AValue: Double): Double;

function CreateConverter(const AFactor: Double): TConverterFunc;
begin
  Result := function (const AValue: Double): Double
            begin
              Result := AValue * AFactor;
            end;
end;

procedure TestAnonMethod;
var
  InchesToCentimetres, CentimetresToInches: TConverterFunc;
  Value: Double;
begin
  InchesToCentimetres := CreateConverter(2.54);
  CentimetresToInches := CreateConverter(1 / 2.54);

  Value := InchesToCentimetres(100);
  WriteLn(Format('100" = %n cm', [Value]));

  Value := CentimetresToInches(100);
  WriteLn(Format('100 cm = %.2n"', [Value]));
end;
```

The previous three examples all have the constructor function taking a parameter. Of course, that doesn't have to be the case. For example, the following program uses an anonymous method as a simple 'iterator', producing an ordered series of Fibonacci numbers:

```
uses System.SysUtils;

function CreateFibonacciIterator: TFunc<Integer>;
var
  I, J: Integer;
begin
  I := 0; J := 1;
  Result := function : Integer
            begin
              Result := I;
              I := J;
              Inc(J, Result);
            end;
end;

var
  Iterator: TFunc<Integer>;
  Num: Integer;
begin
  Iterator := CreateFibonacciIterator();
  repeat
    Num := Iterator;
    WriteLn(Num);
  until Num > 10000;
end.
```

When your constructor function doesn't take any arguments, one thing you'll find is the compiler being picky over brackets. In particular, you will need to use an empty pair of brackets when assigning the constructed method to a variable (`Iterator := CreateFibonacciIterator();`). These tell the compiler you definitely intend to get a reference to the function returned (in this case, an instance of `TFunc<Integer>`), rather than the result of that function (an `Integer`).

Anonymous methods under the bonnet

This section began by observing how anonymous methods do not seem to be methods when they are anonymous, and are not necessarily anonymous given their variables can be assigned ordinary named methods too. Nevertheless, we have seen how dedicated constructor functions can be written that roughly correspond to the constructors you might define for classes. Moreover, we have found anonymous methods to 'capture' variables in a way that makes those variables a close analogue to the fields of a class.

There is a good reason for this: behind the scenes, an anonymous method's captured variables *are* fields of a class, namely of one descended from `TInterfacedObject`. This class is defined by the compiler to represent the anonymous method, the method's code becoming the code for an actual method called `Invoke`. The method reference type implemented by the anonymous method is then mapped to an interface implemented by the hidden class, with the lifetime of the anonymous method being governed by normal `IInterface` reference counting. (In the case of an ordinary named method being assigned to a method reference variable, a wrapper object is created.)

So, consider the following method reference type:

```
type
  TMyFunc = reference to function (const S: string): Boolean;
```

This is technically equivalent to the following interface type:

```
type
  IMyFunc = interface
    function Invoke(const S: string): Boolean;
  end;
```

On assigning an anonymous method to a variable typed to `TMyFunc`, you are therefore creating and assigning an object that implements `IMyFunc`.

Generally speaking, you shouldn't rely on anonymous methods being implemented like this, just as you shouldn't rely on implementation details generally. Nonetheless, the basic implementation is easy enough to prove in code:

```
uses System.SysUtils;

```

```
function CreateAnonProc: TProc;
begin
  Result := procedure
            begin
              WriteLn('Hiya');
            end;
end;

function UnsafeIntfCast(const Source): IInterface;
begin
  Result := IInterface(Source);
end;

var
  Proc: TProc;
  Intf: IInterface;
  Cls: TClass;
begin
  Proc := CreateAnonProc();
  Intf := UnsafeIntfCast(Proc);
  Cls := TObject(Intf).ClassType;
  repeat
    WriteLn(Cls.ClassName);
    Cls := Cls.ClassParent;
  until Cls = nil;
end.
```

This produces the following output:

```
CreateAnonProc$0$ActRec
TInterfacedObject
TObject
```

Anonymous methods as classes

The fact method reference types are really a special sort of interface type suggests it should be possible for them to be implemented by classes you define explicitly — and that possibility turns out to be a genuine one:

```
type
  IMyAnonMethodType = reference to procedure (const S: string);

  TMyAnonMethodImpl = class(TInterfacedObject, IMyAnonMethodType)
    procedure Invoke(const S: string);
  end;

procedure TMyAnonMethodImpl.Invoke(const S: string);
begin
  WriteLn(S);
end;

var
  Proc: IMyAnonMethodType;
begin
  Proc := TMyAnonMethodImpl.Create();
  Proc('A bit weird, but works fine');
end.
```

A practical use of this is to implement functionality similar to ‘smart pointers’ in C++. A ‘smart pointer’ is a sort of transparent wrapper round an underlying object that frees the object when the wrapper itself goes out of scope:

```
type
  ISmartReference<T: class> = reference to function: T;

  TSmartReference<T: class> = class(TInterfacedObject,
    ISmartReference<T>)
  strict private
    FObject: T;
  protected
    function Invoke: T;
  public
    constructor Create(AObject: T);
    destructor Destroy; override;
  end;

constructor TSmartReference<T>.Create(AObject: T);
begin
  inherited Create;
  FObject := AObject;
end;

destructor TSmartReference<T>.Destroy;
begin
  FObject.Free;
  inherited;
end;

function TSmartReference<T>.Invoke: T;
begin
  Result := FObject;
end;
```

Credit for this implementation goes to Barry Kelly, the compiler engineer at Embarcadero who actually implemented anonymous methods in the Delphi compiler (`http://blog.barrkel.com/2008_11_01_archive.html`). Here's it put to work:

```
type
  TTest = class
    Data: string;
    destructor Destroy; override;
  end;

destructor TTest.Destroy;
begin
  WriteLn('TTest.Free');
  inherited;
end;

procedure WorkWithTestObj(Obj: TTest);
begin
  WriteLn(Obj.Data);
end;

procedure DoTest;
var
  TestObj: ISmartReference<TTest>;
begin
  TestObj := TSmartReference<TTest>.Create(TTest.Create);
  TestObj.Data := 'Anonymous methods can be pretty neat';
  WorkWithTestObj(TestObj);
end;

begin
  DoTest;
end.
```

‘Advanced’ records: operator overloading

Traditionally, record types in Pascal were very simple things, containing public fields and nothing else. As we saw in the previous chapter, Delphi adds the ability to include methods, visibility specifiers and static data however, producing something more akin to a stack-based object type. All those additional features are shared with classes, but as classes offer functionality that records lack, so records offer functionality that classes lack: namely, operator overloading.

‘Operator overloading’ is when a record type is made to work with one or more operators such as `+` and `=`, together with one or more standard procedures such as `Inc` and `Dec` — without it, the only operator a record supports natively is assignment (`:=`). Exploit this facility, and you can create new basic types that appear to work just like the basic types built into the language.

Syntax

For each operator or standard procedure you wish to support, one or more special methods need to be defined on the record type. These have the following general form:

```
class operator OperatorName(const A: TypeName1
  [; const B: TypeName2]): TypeName3;
```

`OperatorName` should be replaced with one of the following identifiers. In each case, the operator or standard routine it pertains to is in brackets:

- *Binary and string operators:* `Add` (`+`), `Subtract` (`-`), `Multiply` (`*`), `Divide` (`/`), `IntDivide` (`div`), `Modulus` (`mod`), `LeftShift` (`shl`), `RightShift` (`shr`), `LogicalAnd` (`and`), `LogicalOr` (`or`), `LogicalXor` (`xor`), `BitwiseAnd` (`and`), `BitwiseOr` (`or`), `BitwiseXor` (`xor`). These all take two parameters, one of which must be typed to the record itself. The result can be typed similarly, but it doesn’t have to be. E.g., you could have an `Add` implementation that concatenates two record instances together and returns a string.
- *Comparison operators:* `Equal` (`=`), `NotEqual` (`<>`), `GreaterThan` (`>`), `GreaterThanOrEqual` (`>=`), `LessThan` (`<`), `LessThanOrEqual` (`<=`). Each of these takes two parameters and returns a `Boolean`.
- *Set-specific operators and standard routines (these don’t have to be used with sets and quasi-set types):* `Include` (`Include`), `Exclude` (`Exclude`), `In` (`in`). As with the comparison operator functions, `In` must return a `Boolean`.
- *Unary operators:* `Negative` (`-`), `Positive` (`+`), `LogicalNot` (`not`), `Inc` (`Inc`), `Dec` (`Dec`), `Trunc` (`Trunc`), `Round` (`Round`). Each of these takes a single parameter, and returns a new instance of your record type.

All class operator methods are implemented as functions. This is so even if the method pertains to a standard routine that takes the form of a procedure with a `var` parameter, such as `Inc` or `Dec`. For example, here’s a custom implementation of `Inc`:

```
uses
  System.SysUtils;

type
  TMyRec = record
    Data: Double;
    class operator Inc(const Source: TMyRec): TMyRec;
  end;

class operator TMyDoubleRec.Inc(const Source: TMyRec): TMyRec;
begin
  Result.Data := Source.Data + 1;
end;

var
  Rec: TMyDoubleRec;
begin
  Rec.Data := 99.9;
  Inc(Rec);
  WriteLn(FloatToStr(Rec.Data)); //100.9
end.
```

That’s a pretty trivial example, so here’s a slightly more complicated one for `Include`. In it, the `Include` standard routine (usually used to add an element to a set) is repurposed to add a callback (in this case an anonymous method) to a list of callbacks maintained by the record:

```
type
  TMyCallback = reference to procedure (const AData: string);
  TMyStrRec = record
  strict private
    FCallbacks: TArray<TMyCallback>;
  public
    Data: string;
    class operator Include(const ASource: TMyStrRec;
      const ACallback: TMyCallback): TMyStrRec;
    procedure ExecuteCallbacks;
  end;

class operator TMyStrRec.Include(const ASource: TMyStrRec;
  const ACallback: TMyCallback): TMyStrRec;
var
  M: Integer;
begin
  Result := ASource;
  M := Length(Result.FCallbacks);
  SetLength(Result.FCallbacks, M + 1);
  Result.FCallbacks[M] := ACallback;
end;

procedure TMyStrRec.ExecuteCallbacks;
var
  Callback: TMyCallback;
begin
  for Callback in FCallbacks do
    Callback(Data);
end;

var
  Rec: TMyStrRec;
begin
  Rec.Data := 'Operator overloading is not just for operators!';
  Include(Rec,
    procedure (const AData: string)
    begin
      WriteLn(AData);
    end);
  Rec.ExecuteCallbacks;
end.
```

When implementing `Include`, both the first parameter and the result must be typed to the record. The type of second parameter — the thing being ‘included’ — is immaterial.

Implementing assignments or typecasts from or to a record type

In addition to the operators previously listed, an `Implicit` ‘operator’ is available to implement simple assignments to or from your record type, and `Explicit` to allow explicit typecasts to or from it. Both these ‘operators’ can be overloaded in order to support multiple destination or target types:

```
uses System.SysUtils;

type
  TMyIntRec = record
  strict private
    Data: Integer;
  public
    //enable assigning a TMyIntRec to an Integer
    class operator Implicit(
      const Source: TMyIntRec): Integer; overload;
    //enable assigning a TMyIntRec to a string
    class operator Implicit(
      const Source: TMyIntRec): string; overload;
    //enable assigning an Integer to a TMyIntRec
    class operator Implicit(
      const Source: Integer): TMyIntRec; overload;
  end;

class operator TMyIntRec.Implicit(
  const Source: TMyIntRec): Integer;
begin
  Result := Source.Data;
end;

class operator TMyIntRec.Implicit(
  const Source: TMyIntRec): string;
```

```
begin
  Result := IntToStr(Source.Data);
end;

class operator TMyIntRec.Implicit(
  const Source: Integer): TMyIntRec;
begin
  Result.Data := Source;
end;

var
  Rec: TMyIntRec;
  S: string;
begin
  Rec := 123;
  S := 'The data is ' + Rec;
  WriteLn(S);
end.
```

Unfortunately, one thing you *can't* provide a custom implementation for is the assignment of one instance of your record to another. In other words, it is not possible to define an `Implicit` overload that both receives and returns an instance of the same type. This means that assignments between the same record type will *always* involve a simple copy of the first record's fields to the second.

A related limitation of the 'advanced' record syntax is that you cannot control what happens when an instance of your record is first 'created'. While records support the constructor syntax, this is only for custom constructors that take one or more parameters. If and when such a constructor is defined, it merely supplements rather than supplants the implicit default 'constructor' that takes no parameters.

This will be problematic whenever you wish to implement a type that requires some sort of custom initialisation. You can't just add a private Boolean field to act as a flag, since in many cases the field itself won't be initialised to `False`:

```
type
  TMyBadRec = record
  strict private
    FInitialized: Integer;
  public
    //...
    property Initialized: Boolean read FInitialized;
  end;

procedure Test;
var
  Rec: TMyBadRec;
begin
  WriteLn(Rec.Initialized); //may output TRUE, or may not...
end;
```

The workaround is to declare a field with a 'managed' type to act as a flag, and check whether this flag remains `nil` (or in case of a `string`, empty) whenever something substantive happens. This will work since the instances of a managed type will *always* be zero-initialised.

Example: a string type with a minimum and maximum length

As an extended example of an 'advanced' record type, let us implement a special sort of string (`TBoundedString`) that enforces a minimum and maximum length. The valid bounds will be specified in a constructor, and retrievable via properties. If an attempt is made to use an instance without explicitly constructing it first, an exception will be raised.

Breaking the basic requirements down, the following will need to be implemented:

- An overloaded constructor to specify the bounds and (optionally) the initial string data.
- Enforcement of the bounds specified in the constructor.
- A `Value` property to hold the string data, and `MinLength` and `MaxLength` properties to report the bounds.
- Simple assignments to a string without the need for an explicit cast or function call.
- Use of the string concatenation operator (+) with one or more `TBoundedString` instances.
- A runtime exception on any attempt to read a property's value without having explicitly created the `TBoundedString` first.

Ideally, you would be able to directly assign an ordinary string to a `TBoundedString` as well as directly retrieve string data.

However, this isn't possible due to assignments being a matter of *replacing*, not amending the old instance. This wouldn't matter if the new instance could discover the bounds of the old one — but it can't. As a result, the string data of an existing `TBoundedString` instance will only be changeable by assigning the `Value` property.

Implementing TBoundedString

In essence, `TBoundedString` will be wrapping a normal string. The `Value` property, therefore, can simply read and set this string after some error checking is performed. If we say an initialised `TBoundedString` must have at least one character, the internal normal string field can also serve a second purpose: if it is empty, then we know the record has not been explicitly constructed.

With that in mind, the interface of `TBoundedString` can look like this. For enforcing the minimum length, a custom sub-range type, `TBoundedString.TLength`, is used:

```
type
  TBoundedString = record
  public type
    TLength = 1..High(Integer);
  strict private
    FMinLength, FMaxLength: TLength;
    FValue: string;
    procedure CheckInitialized;
    function GetMinLength: TLength;
    function GetMaxLength: TLength;
    function GetValue: string;
    procedure SetValue(const S: string);
  public
    constructor Create(AMinLen, AMaxLen: TLength); overload;
    constructor Create(AMinLen, AMaxLen: TLength;
      const AValue: string); overload;
    class operator Add(const A, B: TBoundedString): string; overload;
    class operator Add(const A: TBoundedString;
      const B: string): string; overload;
    class operator Add(const A: string;
      const B: TBoundedString): string; overload;
    class operator Implicit(const S: TBoundedString): string;
    class operator Equal(const A, B: TBoundedString): Boolean;
    class operator NotEqual(const A, B: TBoundedString): Boolean;
    property MinLength: TLength read GetMinLength;
    property MaxLength: TLength read GetMaxLength;
    property Value: string read GetValue write SetValue;
  end;
```

The `Add` operator requires three overloads to cover all the bases — `MyBoundedStr + MyOtherBoundedStr`, `MyBoundedStr + MyNormalStr`, and `MyNormalStr + MyBoundedStr`. Since we are only supporting direct assignment *to* a normal string, we only need the one `Implicit` implementation. A further point about the operator overloads is that since we implement `Equal`, implementing `NotEqual` isn't strictly necessary — in the absence of an explicit `NotEqual` implementation, the compiler would just call `Equal` and negate the result. We will implement `NotEqual` for completeness' sake however.

Here's what the code to do all this looks like:

```
resourcestring
  SNotInitialized = 'A TBoundedString instance requires ' +
    'explicit creation before use';
  SStringTooSmall = 'String too small for bounded string';
  SStringTooBig = 'String too big for bounded string';

constructor TBoundedString.Create(AMinLen, AMaxLen: TLength);
begin
  FMinLength := AMinLen;
  FMaxLength := AMaxLen;
  FValue := StringOfChar(' ', AMinLen);
end;

constructor TBoundedString.Create(AMinLen, AMaxLen: TLength;
  const AValue: string);
begin
  Create(AMinLen, AMaxLen);
  SetValue(AValue);
end;

procedure TBoundedString.CheckInitialized;
begin
  if FValue = '' then
```

```
    raise EInvalidOpException.CreateRes(@SNotInitialized);
end;

function TBoundedString.GetMinLength: TLength;
begin
  CheckInitialized;
  Result := FMinLength;
end;

function TBoundedString.GetMaxLength: TLength;
begin
  CheckInitialized;
  Result := FMaxLength;
end;

function TBoundedString.GetValue: string;
begin
  CheckInitialized;
  Result := FValue;
end;

procedure TBoundedString.SetValue(const S: string);
begin
  CheckInitialized;
  if Length(S) < FMinLength then
    raise ERangeError.CreateRes(@SStringTooSmall);
  if Length(S) > FMaxLength then
    raise ERangeError.CreateRes(@SStringTooSmall);
  FValue := S;
end;

class operator TBoundedString.Add(const A, B: TBoundedString): string;
begin
  Result := A.Value + B.Value;
end;

class operator TBoundedString.Add(const A: TBoundedString;
  const B: string): string;
begin
  Result := A.Value + B;
end;

class operator TBoundedString.Add(const A: string;
  const B: TBoundedString): string;
begin
  Result := A + B.Value;
end;

class operator TBoundedString.Equal(const A, B: TBoundedString): Boolean;
begin
  Result := A.Value = B.Value;
end;

class operator TBoundedString.NotEqual(const A, B: TBoundedString): Boolean;
begin
  Result := A.Value <> B.Value;
end;

class operator TBoundedString.Implicit(const S: TBoundedString): string;
begin
  Result := S.Value;
end;
```

Here it is in use, testing its various features:

```
uses
  System.SysUtils, BoundedStrings;

var
  Max5Str, AtLeast2Max20Str: TBoundedString;
begin
  try
    Max5Str := TBoundedString.Create(1, 5, 'Hello');
    AtLeast2Max20Str := TBoundedString.Create(2, 20,
      Max5Str + ' world');
    WriteLn('Max5Str must have at least ', Max5Str.MinLength,
      ' characters and at most ', Max5Str.MaxLength);
    WriteLn('Max5Str = ' + Max5Str);
    WriteLn('AtLeast2Max20Str = ' + AtLeast2Max20Str);
    { Should be OK, because within bounds }
    Max5Str.Value := 'Bye';
    AtLeast2Max20Str.Value := 'cruel world';
    { Test using string concatenation operator (i.e., +) }
    WriteLn(Max5Str + ' ' + AtLeast2Max20Str);
    { Test inequality and equality operators, and sending a
      TBoundedString to a function expecting a normal string }
    if Max5Str <> 'bye' then
      if SameText(Max5Str, 'bye') then
        if Max5Str = 'Bye' then WriteLn('Looking good!');
    { Test the actual bounds functionality! }
    WriteLn('Hopefully an exception will now be raised...');
    Max5Str.Value := 'I said goodbye!';
  except
    on E: Exception do
      WriteLn(E.ClassName, ': ', E.Message);
  end;
end.
```

Go forth and multiply?

If you try the code out, you should find it working pretty nicely. However, does that mean you should you go ahead and create lots of custom base types? Possibly not, since achieving the *appearance* of a built-in type is not the same as actually *implementing* one. In the present example, for instance, concatenating multiple `TBoundedString` instances (`S1 + S2 + S3 + S4`) will never be as fast as concatenating multiple `string` instances. This is because it will involve the low-level RTL's generic record copying code, which is relatively slow. Nonetheless, in moderation — and where you aren't simply attempting to reinvent the wheel — operator overloading can be a powerful tool in the toolbox.